tion, if a lot of physical memory is desired, a manual cache approach may be better (the better packing should overcome any loss due to redundancy with the OS disk cache). Compression of page sets invokes several issues that probably can't be fully addressed until after 1.0. **Bottom line: the target now for 1.0 is to use per-file-mmapping with per-page compression (but no compression of page sets)**. Also, we should instrument the system to allow us to easily collect the relevant data (mem usage, CPU load of different routines, etc.) to help guide us in further evolution of the system to improve performance (e.g. compressed page sets or explicit page cache).

CPU

The main work of the CPU is as follows (encryption is assumed to be done by hardware since its CPU impact is severe):

1. OS system call to retrieve request from network.
2. Decode client request.
3. Validate AccessToken.
4. Lookup AppID, File # in primary lookup hash table. (If mmapping eStream sets, instead lookup in App table, then lookup in FOST).
5. If mmapping then (if uncompressed, no further lookup, if compressed, then lookup in POST to find page and size), if explicit cache then look in B-tree (secondary lookup).
6. If lookup fails, then bring in the data off disk (either mmap or file system call).
7. Copy page data to reply buffer.
8. OS system call to send reply to network.

However, if compressed page sets are used, lookups get more complicated, with a different set of tables to check for an appropriate page set first (and lookup failures incur potential decompression/compression). It appears the least amount of CPU time is probably incurred when doing per-file-mmapping. All pages held in memory are kept in compressed form to save repeated compression of the same data, so pretty much all the work is in lookups and memory copies. Potentially the AccessToken validation will use hardware assist. Lookup failures (i.e. having to go to disk) should be relatively uncommon, and memory should be sized to ensure that.

However, since the AS will run in user mode, this incurs the penalty of two extra copies (from the network buffers) and switching between kernel and user mode twice. If this is enough of a problem, we'll have to consider implementing the AS to run in the kernel (all commercial NFS, etc. implementations run in the kernel), which means we should choose our implementation to be compatible with that approach. In particular, we may not be able to rely on the virtual address space not being fragmented, so mmapping full eStream sets may be impossible. Plus robustness of the server becomes even more important, and portability issues arise. For the 1.0 release, we plan to implement the AS in user mode keeping the possibility of moving to kernel mode in the future, and will collect data from 1.0 (or derived prototype) to evaluate the actual benefits.

Disk: Since we are relying heavily on the common pages being in memory, we could possibly even consider storing the processed sets on a network disk, i.e. remote from the app server itself. However, such sharing won't work well for compressed page sets since

they are written to at runtime—it would be extremely messy to handle dozens of app servers trying to add many compressed page sets (possibly the same) to a set of shared files.

Multithreading model

The approach will be to have a single boss thread which pulls things out of the network port and stuffs client requests into a queue and a bunch of worker threads which grab requests and send back the replies. Simple enough, but this raises the issue of thread control, since the boss also needs to be able to handle threads that die or hang and kill and restart them. The boss thread will monitor the worker threads and provide load/hearbeat info to the monitor through the server manager thread, thus giving visibility to the server monitor of the health of all the worker threads.

Load balancing

To be described elsewhere? (appears in SLiM server LLD)

Security

There are two levels of security involved in the AS. First, we must prevent clients who don't hold valid licenses from gaining access to the licensed binaries. This is accomplished by the client obtaining an AccessToken from the SLiM server and presenting it to the AS upon every request. The AS can then use the SLiM server's public key to test the authenticity of the AccessToken (to protect against forgeries), and then can test the authentic expiration time of the AccessToken. Second, we must encrypt the actual data being sent on the wire to prevent third parties from gathering the binary data covered by the license. Since the data coming out is somewhat obfuscated anyway (files are identified by arbitrary IDs, with our own strange message formats and compression and all in random pieces, etc.) it is not clear how much extra protection is really necessary, i.e. what do the license issuers actually want? We should use a common scheme like SSL to perform this encryption. It has been decided that the encryption load for this would be too great, and thus the data send back will be unencrypted. We may use SSL for authentication purposes only (i.e. null-cipher), if that is cheap enough.

Also, a possible optimization for checking AccessTokens would be to cache recently used AccessTokens along with a signature/hash. If a token presented by a client matches, then we can skip the authentication step (since we've done it once already) and just check the expiration time.

Robustness

The AS must be very robust. It must catch OS call errors and handle/log them as appropriate, and deal with threads that hang or die. Thus it needs to aggressively check for error conditions and possible failure modes. The AS also needs to track relevant resources (e.g. sockets, memory) and carefully manage/reclaim them so as not to exceed any limits or to degrade performance. And of course, the AS needs to check all data coming in from the client, to deal with ill-formed requests, and illegal values (e.g. huge negative indexes, etc.), and perform no potentially dangerous operation without validating parameters. This becomes even more important when we eventually move the AS to run in kernel mode. The AS also needs to be as stateless as possible, to minimize recovery time, and if it does perform writes to disk (such as for the compressed page sets), do so in a reliable fashion conducive to quick recovery. Any unreliability in the AppServer will negate any benefit of scalability we have over our competitors.

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

The various components of the AS are not too large or complicated: The request dispatcher (to worker threads), the hash table, the compression code, the AccessToken checking code, etc. These shouldn't be too hard to do reasonable testing on in isolation.

For the post-processor component, we'll have to build some sample Estream Sets as input, but it'll be hard to tell whether the output is correct without having a minimal working AS.

Cross-component testing plans

The best approach will be to perform incremental implementation and testing. I.e. we build the core functionality that is required (i.e. can start with just regular i/o reads), and then add the more performance-related stuff later (adding mmaping, and then the hash table & AccessToken checks), while testing the entire system as pieces are gradually added (of course performing sanity-check and other minimal testing on the pieces first if possible). Compression can be added last.

To actually drive the AS, we'll need a test client, which will be designed to just shoot off a series of read requests to the server. The file data returned could then be written to files, and this can be compared against the original set of files used to create the Estream Set we started with, to check that the data was received properly. For checking error conditions, a log of errors can be written and compared against a reference log for those requests we expect to fail.

Stress testing plans

To accomplish this, we should run multiple independent test clients (on the same machine and on different machines), and increase the frequency of requests (to stress the AS's threads and sychronization, and communication routines), and the number and size of files referenced (to stress the hash table and memory). Each test client can then check whether the data and errors it got back were as expected, like in the above subsection.

Coverage testing plans

Should we use some kind of code coverage tool for this?

Performance testing plans

Since performance is critical, we should take the time to evaluate the AS's performance characteristics. We need to crank up our stress testing until either bandwidth or CPU saturates, and record the request rate that generated it. We should compare how this point responds to high numbers of clients with fewer requests per client vs. fewer clients with higher requests per client. We'll need to profile the system to find bottlenecks to tweak more performance out of it, and learn how well our original design assumptions hold up. Depending on whether CPU or bandwidth (or memory) saturates first, we may want to modify the system's tradeoffs to improve scalability further, and otherwise note which components a customer should upgrade for better performance. Also, if we think we can come up with reasonable client access pattern profiles, we may want to use those to estimate the actual number of real-world clients an AS can support. As part of this, we'll probably want to run the AS in-house once it is mature enough (eat our own dogfood), and then farm out app upgrades, etc. (play out some of our scenarios) and see what happens to the AS's (do they choke or what).

Availability testing plans

We will also need to test our failover and load balancing capabilities. This will require several test machines with the monitor in place to start and stop servers, and have clients be aware of multiple AS's and respond appropriately when an AS stops responding. For load balancing, we'll probably want a bunch of test clients with a variety of access patterns and see how well their requests are distributed.

Upgrading/Supportability/Deployment design

App Servers will possibly need to version their interface with clients (requiring clients to state the version they're expecting), but will also need to support older versions. We may also modify the Estream Set format (or just the processed set format), but that should be handled by upgrading both the AS and post processor and then regenerating the processed sets.

For supportability & deployment, the AS will report error conditions and load to the server monitor, which is used by the customer.

Open Issues

1. Is there a limit to the # of possible mmaps?
2. Is there a single system call to unmap all mmaps?

eStream Server Component Framework
Low Level Design

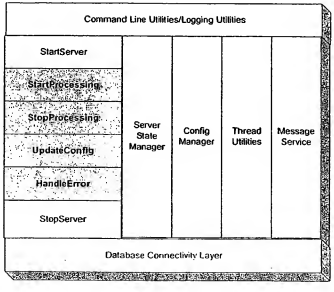
Michael Beckmann

Functionality

The *Server Component Framework* provides a common basis on which server components are implemented. The framework provides a number of services such as common server initialization and configuration, messaging, state management, logging, and error handling. The component framework ties together many of the core utilities provided for the server components.

The advantage of the framework is that heterogeneous server components can be managed in a consistent manner with the expectation that all server components will communicate and behave consistently within the system.

All server components with the exception of the web server will be built on top the *Server Component Framework*. To make use of the *Server Component Framework*, a specialized server component will need to extend the framework by implementing the methods high-lighted in gray. Implementing these interfaces makes the specialized server component "plug-able" within the framework.



The following table give a brief description of each of the routines that need to be specialized by each server component to make it plug-able into the Server Framework:

StartProcessing	Specialized server component routine to request the server component to start processing work.
StopProcessing	Specialized routine to request the server component stop processing work and transition into an idle state
UpdateConfig	Specialized routine to dynamically update configurations while a component is either in the processing or idle state.
HandleError	Specialized routine to handle the occurrence of an error

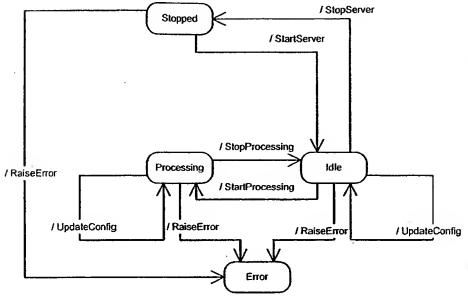
Server State Manager:

At the heart of the server component framework is the *Server State Manager*. The server state manager is a set of interfaces that initiate and manage state changes within a server component. All Server components, by virtue of being built on top of the component framework, can be managed uniformly across a deployment.

The *Server State Manager* implements a simple state machine that is shared between components. It manages the state transitions within the server component. Additionally, the state manager maintains current state information for each server component and logs state transition history in the event that a server component terminates unexpectedly.

As specified above, each server component is required to implement a number of transition methods, with pre-defined signatures, which the state manager will execute when making a state transition.

The following diagram shows the state diagram and the associated transitions:



Message Service:

The *Server Component Framework* depends on a message service which is used by the Server State Manager and Configuration Manager to communicate with the System Monitor.

The *Server State Manager* uses the messaging service to listen for state change requests from the System Monitor which it satisfies by returning the current state, any up-to-date status, and load information.

The *Configuration Manager* uses the message service to request configuration information from the *System Monitor*. Although each server component could easily go to the database for configuration information, it has been decided to go through the monitor as to save db licensing costs.

See below for more details on messaging protocols for the *Server State Manager* and the *Configuration Manager*. Also, refer to the low-level design document for details on the design of the eStream Messaging Service (EMS).

Configuration Management:

The configuration management utility is used by all server components to manage the server configurations. It provides the following functionality:

- Configuration for a server consists of a set of name – value tuples where the values themselves can be a set of name-value tuple.
- Servers can load the complete configuration from the database (indirectly).

- Servers can load the configuration for a given name.
- Servers can load the configuration from a flat file also.

On the Server Manger interface, configuration will appear as a table containing name – value tuples. The table may be hierarchical to represent nested structures containing the values which can themselves be name values. An example of a simple name-value pair would be:

port 8080

An example of nested name values would be:

Applications:
word.exe *windows2000sp3*
excel.exe *win98sp4*

On a flat file the configurations will always be name-value pairs. To represent one level nested structure the format would be:

Applications word.exe windows2000sp3
Applications excel.exe win98sp4

A common set of configurable parameters is defined for all server components. These configurations are maintained by the *Server Component Framework* in collaboration with the *Configuration Manager*. All configuration information is persistently stored within the database. The common configurations are used to initialize the server component after the component process has been launched. Refer to the configuration table below for more details on common configurations. Specialized server components can support additional configurations (non-common) depending on the server type. These configurations are read from the database and updated when a server component starts processing. They can also be updated dynamically while a server component is processing through the use of the **UpdateConfig** interface.

The list of common configurations include:

Information	Supports Dynamic Config	State	Description
ServerID	No		Unique identifier for server components. This server identify is unique within a deployment. This ServerID is not known to eStream clients. Its purpose is as a handle to uniquely identify server components.
ServerType	No		Identifies the type of server component. One of the following applies: ▪ Primary Monitor

			▪ Backup Monitor ▪ Application Server ▪ SLiM Server
DbUser	No		User name string required for database connectivity for this server ID
DbPasswd	No		Database password associated with the DbUser
Dsn	No		Data Source Name used to access the database.
PortNum	No		PortNumber used for light-weight messaging listener
MachineID	No		Machine ID is used to get at important machine information needed for all server components such as: ▪ IP address for the machine server component is hosted on ▪ Domain name for the machine ▪ Machines name
AutoReStart	Yes	Any State	Flag indicating that server component process can be restarted automatically without manual intervention. This info is consumed by the System Monitor.
HeartBeat-TimeOut	Yes	Any State	Specifies the timeout period for the listener. If the timeout period is reached. The component assumes that it has lost the connection. All Server components have a listener by which they receive instructions from the primary system monitor. Even the monitor has a listener that communicates with the Server Admin UI.
HeartBeatRate	Yes	Any State	Frequency at which the heart-beat is sent to this server component. Specified in milliseconds. This item is consumed by the System Monitor.

Command Line Utilities:

The *Command Line Utilities* component provides a consistent way to define and process command line arguments. To use this utility, the using component must define a table of arguments, which defines the valid set of arguments, whether or not they are required, and any default values.

Arguments are specified on the command line as name/value pairs. The utility implements the following command line syntax to support the name/value pairs. The argument syntax is defined as follows:

<name>=<value>

name	Name is an alpha-numeric identifier. The Name can be of arbitrary length as supported by the system however shorter names are recommended. Names are case sensitive
value	Any alpha-numeric value. Punctuation characters may also be used. Values are

	case sensitive

There can be no spaces between the <name>, "=", and the <value> elements. The existence of one or more spaces or tabs delineates separation between arguments on the command line.

Example: server.exe sid=267 dsn=oracle user=michaelb passwd=mypasswd

- If a named argument is specified more than once on the command line, subsequent arguments will cause a diagnostic to be issued and the argument will be ignored.
- This utility allows the user to specify default values for arguments. If a default value is defined then the argument will be processed with its default in the event that the argument is not specified on the command line.
- This utility allows the user to tag specific arguments as required. If the required argument is not specified on the command line this utility will raise a diagnostic for the required argument. Not specifying a required argument will cause a fatal error.

The following options are supported:

sid	Server Component Identifier. Each server component within a deployment is uniquely identified via the sid. The sid is a handle into the database for accessing information unique to a specific server component.
dsn	Data Source Name. A data source name is necessary to establish an ODBC connection. Data Source Names are generated by an ODBC administrative tool
dbuser	User name. For database access security, all components need to connect as a specific user.
dbpasswd	password associate with the dbuser

Logging Utilities:

All servers and clients in eStream 1.0 need to log the error and access data. Logging enables component debugging and auditing support.

EStream Framework should provide logging with the following features:

- Each component will have an error and optioanally an access log file. The names of these files would be <component>_error.log and <component>_access.log.
- The files will be located in the <eStream1.0 Root Dir>\logs directory.
- The error log files will have messages with the following priorities:
  - 4-Low : A warning which can be ignored.
  - 3-Medium: A warning which needs to be looked into.
  - 2-High: Recoverable Error in the component.
  - 1-Critical: Fatal Error. Needs admin assistance.

- Logging level should be configurable. The following levels are to be supported.
  - 0: Only errors will be logged. This will be the default level.
  - 1: Errors and Warnings to be logged.
  - 2: Errors, Warnings and Debugging information to be logged.
  - 3: Errors, Warnings and advanced Debugging (like memory dumps, tcp stack dumps etc) to be logged.
- Log Wrapping to be supported. The log files will wrap at a predefined size. On wrapping the following actions will occur:
  - Any existing <logfile>.bak will be deleted from the system.
  - The current <logfile> will be backed to <logfile>.bak
  - The component will continue logging to the <logfile>.

For each eStream client and server component logging the log files (component_error.log and component_access.log) should be written in eStream1.0Root\logs directory. The formats for the log files will be as follows:

Error Log:

[HEADER]
[TimeStamp] [Thread ID] [Priority] [Message]
...
[FOOTER]

An example of this log format would be:

```
**********************************************
Omnishift eStream Application Server
Server Started.
StartTime: [redacted]:31:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
**********************************************

[[redacted]16:31:19 -0700] 0 2-High Cannot connect to the database.
Invalid Username/Password.
[[redacted]:16:31:19 -0700] 1 1-Critical Cannot start the HTTP listener
at port 80.
[14/Aug/2000:16:31:19 -0700] 0 1-Critical Shutting down the server.

**********************************************
Omnishift eStream Application Server
Server Stopped.
StopTime: [redacted]16:35:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
**********************************************
```

Format of Access Log Message:

[HEADER]
[TimeStamp] [Thread ID] [Message]
[FOOTER]

Data type definitions

Server State:

The server components can be in any one of the following states:

State	Description
STOPPED	If a server is in the STOPPED state then the component process is not running.
IDLE	Server component is up and running. The server has been initialized with the common configuration and the messaging system has been enabled. The listener is actively waiting on the System Monitor for transition requests. The server component is not processing any work specific to this servers specialization.
PROCESSING	Server component is actively taking requests and processing work specific to its specialization. ie. serving access tokens, and application file requests.
ERROR	An error has occurred in the system. Unless the server component is configured with **AutoReStart** and ERROR state must be manually cleared by the server-side administrator.

Server State Transitions:

Changes in server component state are initiated either by the *System Monitor* or directly by the server-side administrator for the system monitor. The exception to this is when an error condition is raised by a server component. In this case, the component will initiate the state change itself. The following state transitions are supported:

Action	Description
START_SERVER	Server is expected to be in the STOPPED state. If a server component is configured to support AutoReStart then the ERROR state is also a valid state from which to initiate this action.
STOP_SERVER	Causes the server to exit its process. The server can be stopped from any state.

START_PROCESSING	Causes the server to change from the IDLE state to the PROCESSING state.
STOP_PROCESSING	Causes the server to change from processing to IDLE state.
UPDATE_CONFIG	Request that the server read its configuration from the configuration manager and change its configuration.
RAISE_ERROR	Request that the server go to ERROR state. This causes an error handler to be called. If the error is fatal it will cause immediate termination of the server process.

Finite State Table:

```
FSMTableEntry ServerStateMgr::FSMTable[] =
{
        { START,{{START_SERVER,STOPPED, START_SERVER,NULL},
                {START_PROCESSING, STOPPED ,START_PROCESSING, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {STOPPED, {{START_SERVER, IDLE, NULL_REQUEST, &StartServer},
                {START_PROCESSING, IDLE, START_PROCESSING, &StartServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {IDLE,{{START_PROCESSING, PROCESSING, NULL_REQUEST,
&StartProcessing},
                {STOP_SERVER, STOPPED, NULL_REQUEST, &StopServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {UPDATE_CONFIG, IDLE, NULL_REQUEST, &UpdateConfig},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {PROCESSING,{{STOP_PROCESSING, IDLE, NULL_REQUEST,
&StopProcessing},
                {UPDATE_CONFIG,PROCESSING, NULL_REQUEST,
&UpdateConfig},
                {STOP_SERVER, IDLE, STOP_SERVER, &StopProcessing},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        { ERROR,{{STOP_SERVER, STOPPED,NULL_REQUEST, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} }

        {NULL_STATE,{{NULL_REQUEST, NULL_STATE, NULL_REQUEST,
        NULL} } }
};
```

Messaging Service Protocol:

A light-weight messaging protocol is needed to facilitate communication between server components. The primary purpose of the messaging protocol is to communicate transition requests to the server components. In response, server components communicate state, status, and load information back to the *System Monitor*.

The messaging protocol supports two primary message types. 1) Requests for the *System Monitor* to perform on other servers. 2) Requests to the server components themselves. These message types are distinguished through the protocol as described below. If the receiver ID and the target ID are identical then the request is for the receiver. If the target is different than the receiver, the message is for the *System Monitor* to enact a request on the target component.

All requests are required to be acknowledged. Without an acknowledgement the message is considered un-received.

OpCode	senderID	receiverID	targetID	Data

The following table describes the protocol used by the Server State Manager in its communication with the System Monitor.

OpCode	Description	Data
0x01	Request for current state	None
0x02	Acknowledgment	▪ Current state ▪ Load info ▪ Status info
0x03	Stop Server request. Acknowledged with 0x02 message	None
0x04	Start Server request. Only valid for System Monitor. Acknowledged with 0x02	None
0x05	Start Processing Request. Acknowledged with 0x02	None
0x06	Stop Processing Request. Acknowledged with 0x02	None
0x07	Update Configuration Request. This is a request for a server component to request its specialized configuration information from the System Monitor and update itself. Acknowledged with 0x02.	None

Interface definitions

Server State Manager:

```
class ServerStateMgr
{
```

```
private:
        ServerState CurrentState;
        static FSMTableEntry FSMTable[];

public:
        ServerStateMgr(void);
        ~ServerStateMgr(void);

        ServerState SetState(ServerState);
        ServerState GetState(void);
        ServerState ProcessRequest(ServerRequest);
};
```

SetState	**Description:** Sets the current state of the server component. 1. Log the state change request 2. Update the state field within the server component in memory data structures. 3. Send message to requester informing them of the successful state change. Note: **SetState** does not update the database directly as in the orginal design. The database is updated by the *System Monitor* once it has received an acknowledgement. A state transition is not complete until **SetState** returns successfully and the Monitor has update the database. **Input:** state value to set current state to. **Output:** current state after the new value has been set. If an error occurs will go to error state. **Errors:** 1. Invalid state argument 2. Failure to either connect or commit state change to the database.

GetState	**Description:** returns the current state. This function does not read from the database to get the current state. The assumption is that if the server component is up and running and that it maintains a valid state. **Input:** none. **Output:** returns the current state. **Errors:** None. Will always return a valid state.

ProcessRequest	**Description:** request to the Server State Manager to change server state. This routine implements the guts of the state machine. 1. Get the current state, and transition request 2. Index into the FSM table and continue to transition from state to state until the transition request is satisfied. 3. Each state transition calls the specialized transition routines for each component. 4. Call to **SetState** to complete each state transition. 5. In the case of an error the state machine will process a RAISE_ERROR request which will call the specialized **Han-**

	dleError and transition to the ERROR state. **Input:** server transition request. Refer to table of valid requests defined above. **Output:** current state after the request has been completed. **Errors:**

Server Component Framework:

```
class ServerComponent: ServerStateMgr{    // abstract base class
private:
        ErrorInfo*      Error; // maintains error if error was detected
        ServerConfig*   Config;  // holds common configuration
        Connection*     Listener; // messaging        utility
public:
        virtual int StartServer(void); // may be specialized by a server component
        virtual int StopServer(void); // may be specialized
        virtual int StartProcessing(void) = 0; // must be specialized
        virtual int StopProcessing(void) = 0; // must be specialized
        virtual int UpdateConfig(void) = 0; // must be specialized
        virtual int HandleError(void) = 0; // must be specialized
        void Run(Request);
}
```

StartServer	**Description:** Called by the *Server State Manager* when a server component is to be started. The **StartServer** routine is provided as part of the SeverComponent class. It performs the following: 1. Send request to System Monitor to request an update of common configuration information. 2. Apply the configuration information to the server component. 3. Construct a listener connection object and start the message service. 4. Return success or failure. Note: ▪ This routine must return immediately to the main thread. Otherwise the *Server State Manager* will be blocked. ▪ Successful return from the **StartServer** routine will put the server into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error condition

StopServer	**Description:** Called by the *Server State Manager*. 1. Perform any necessary cleanup. 2. Send last acknowledgment confirming shutdown to requester 3. Shut down the messaging system and the listener. 4. exit process Note: The monitor will update the database and perform logging.

	Input: None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error value

StartProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by each specialized server component. This routine is used to provide all functionality unique to different types of servers. 1. Spawn a primary processing thread (also known as the boss thread). a. Read server specific configurations unique to this type of server component from the System Monitor b. Spawn worker threads. Depending on the server type this routine does the heavy lifting to either process access tokens and renewals in the case of SLiM server, or process file requests for application servers, or manage and monitor the server components in the case of the *System Monitor*. Note: ▪ This routine must return immediately so that the *Server State Manager* can continue to operate in the main thread. ▪ This routine may make use of the *Server Configuration Manager* for obtaining specialized configuration information **Input:** None **Output:** Value of 0 if successful else error condition. **Errors:** TBD

StopProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by the specialized server component type. 1. Reverse all actions performed by the **StartProcessing** routine. All worker threads should be joined or pooled in waiting state. Successful return from this routine will put the server component into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

UpdateConfig	**Description:** Called by the *Server State Manager*. This routine must be defined by the specific server component type. The purpose of this routine is apply dynamic configurations or update specialized configurations that are unique to this server component. <may require adding a new state to separate dynamic and static configurations> **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

HandleError	**Description:** Component defined error handling routine to handle errors such as timeouts, etc. This routine will need to handle a number of error cases as are possible by the specialized component. The error information is maintained with the ServerComponent class. **Input:** None. **Output:** Integer value designating a handled error or failure. If the error cannot be handled then it is fatal. **Errors:** TBD

Run	**Description:** This routine implements the main processing loop for the server component and runs in the main thread. This routine drives the server component by initiating state requests from the *System Monitor*. Note: The *Server State Manager* always runs in the main thread. 1. Call **ProcessRequest** to transition the server component into the initially requested state. 2. Enter main processing loop a. Check for requests from the message service. b. Call **ProcessRequest** to service the request. c. Send acknowledgement for the request to the message service. Acknowledgement includes new state, load info, and status. **Input:** Initial Transition Request **Output:** None. This routine should never return **Errors:** None.

Server Component Main Loop:

The following main loop is common to all server components:

```
void ServerComponent::Run(ServerRequest Request)
{
        ProcessRequest(Request);
        while (1)
        {
                Request = Listener->GetRequest();
                ProcessRequest(Request);
                Listener->AckRequest(Request, GetState, GetLoad, GetStatus);

        }
}
```

```
#include "ServerArgs.h"
#include "Server.h"
```

```
int main(int argc, char* argv[]) {
        Args = new ArgList();
        Args->ProcessArgList(argv, argc);
        Server = new ServerComponent(GetValue(SID),
                                     GetValue(DSN),
                                     GetValue(DBUSER),
                                     GetValue(PASSWD));
        Server->Run(START_PROCESSING);
}
```

Command Line Utilities:

```
class NameValuePair
{
        private:
                char* Name;
                char* Value;
        public:
                NameValuePair();
                ~NameValuePair();
                char* GetValue(void);
                char* GetName(void);
                char* SetName(char*);
                char*SetValue(char*);
};
```

```
typedef int (*pFunc)(NameValuePair*);

struct ArgTblEntry
{
        char*   Name;
        bool    Required;
        char*   DefaultValue;
        pFunc   ProcessFunction;
};
```

```
ArgTblEntry const ServerArgsTbl[] = {
        {"sid",         true,   0,      &ProcessSId},
        {"dsn",         true,   0,      &ProcessDsn},
        {"dbuser",      true,   0,      &ProcessDbUser},
        {"dbpasswd",    true,   0,      &ProcessDbPasswd},
        {0,             0,      0,      0}
};
```

```
typedef vector<NameValuePair*>    ArgVector;
```

```
class ArgList
{
        private:
                ArgVector                   ArgVec;
                const ArgTblEntry*          ArgTbl;

        private:
                NameValuePair*              ParseArg(char* Arg);
                char*                       ParseName(char* Arg);
                char*                       ParseValue(char* Arg);
                int                         ProcessArg(NameValuePair*);
                int                         FinalizeArgs(void);

        public:
                                            ArgList(const ArgTblEntry*);
                int                         ProcessArgList(char* argv[], int argc);
};
```

ProcessArgList	**Description:** Process the entire argument list. In a loop for each argument argv[] … 1. Call **ParseArg** passing in argv[]. 2. **ParseArg** passes the result to **ProcessArg** 3. After processing the entire argument list and exiting the loop call **FinalizeArgs** **Input:** argv and argc as passed into main() entry point **Output:** integer value designating success or failure **Error:**
ParseArg	**Description:** Takes a char* argument and verifies that it follows that name/value syntax defined as <name>=<value> **Input:** Next char* argument on the list **Output:** **NameValuePair**. NULL will be returned in the event of a syntax error **Error:**
ProcessArg	**Description:** This routine performs the semantic analysis of an argument. 1. Look up name in the **ArgTbl** 2. Verify that the value is valid 3. Add the name value pair to a list of processed arguments called **ArgVec** list. 4. If this name value pair already exists in the list then issue a diagnostic. 5. Call the supplied processing function for this argument as specified in the **ArgTbl** **Input:** **NameValuePair** **Output:** Interger value designating success or failure (0 for success, positive integer for other errors) **Error:**
ParseName	**Description:** Verify that the Name part of the argument conforms to being alpha-numeric **Input:** char* Name part of argument **Output:** char* Name else NULL **Error:** None
ParseValue	**Description:** Verify that the Value part of the argument conforms to being alpha-numeric and/or punctuation characters **Input:** char* Value part of argument **Output:** char* Value else NULL **Error:** None
FinalizeArgs	**Description:** Post process the argument list. The purpose of this routine is to validate that all required arguments have been defined on the command line. Also processes and adds default arguments to the **ArgVec**. **Input:** None **Output:** Success or Failure **Error:**

Configuration Manager:

```
class  Tuple {
        string name;
        Value value;
};

class Value {
        int type;
};

class StringValue: public Value{
        string value;
};

class TupleValue: public Value {
        vector <tuple> tupleArray;
};

typedef vector < tuple > ConfigArray;

class ServerConfig {
private:
        ConfigArray Array;
public:
        ServerConfig(serverId, dsn, dbuser, dbpasswd); // Initialize from db
        ServerConfig(serverId, string filename); // To initialize from a file.

        ConfigArray* GetConfigArray(int serverId);
        Tuple* FindConfig(string Name);
        int Reload(void);
        Tuple* GetConfig(int serverId ,string Name);
};
```

ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes configuration manager. 2. Opens the database and gets configuration array **Input:** Server Id, Data Source Name, Database User name, and database users password. **Output:** None **Errors:**
ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes Configuration Manager. 2. Opens configuration file and reads configuration array. Input: filename of flat-file configuration. **Output:** None **Errors:**

GetConfigArray	**Description:** Returns the entire configuration for a given server id. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying which server to retrieve configuration for **Output:** Returns a vector holding the configuration or NULL **Errors:**

GetConfig	**Description:** Returns the configuration for the specified name. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying the server to retrieve configuration for and Name of configuration item. **Output:** Configuration Tuple. A Tuple may be a nested Tuple. NULL if an error is encountered. **Errors:**

FindConfig	**Description:** Returns the Tuple specified by the name. This routine does not go to the database or flat-file to get its value. Rather it finds the value in the ConfigArray maintained by the Configuration Manager. **Input:** Name of the configuration item. **Output:** Configuration Tuple. NULL if an error is encountered or the Tuple does not exist in the current configuration. **Errors:**

Reload	**Description:** Reloads the entire configuration from the database or flat-file. This routine may reload its configuration indirectly through the use of the System Monitor. In this case it will make a message request to the monitor and listen for the configuration results. **Input:** None **Output:** integer specifying success or failure. Zero will be returned in the case of Success. A negative value in case of error. **Errors:**

Logging Utilities:

```
class LogManager
{
private:
        char* FileName;
        int MaxFileSize;
        char* ResourceFile; // message catalog file

        char* GetMessage(MsgNum, MsgStr)
public:
        LogManager(ServerId,Size=10);
        LogMessage(MsgStr);
        LogMessage(ThreadId, MsgNum, MsgStr, ...);
```

```
};
```

LogMessage	**Description:** Write message out to log file. There are two forms of LogMessage. The first will write out a message buffer as is (unformatted) bypassing the resource file. The second form will format the message. Both forms of LogMessage always pre-append a time stamp. 1. Lookup message number in the resource file and get message string 2. format the log message using time stamp, thread id, etc. 3. write out message into the log file. **Input:** Thread Id, Message Number, Message String, and variable number of arguments. **Output:** None. **Error:**

GetMessage	**Description:** Routine returns a message string from the resource file for the message number specified. **Input:** Message number, C Locale text string. **Output:** Message string. Either way, Get Message will always pass a return a valid message string by either returning the string from the resource file or by passing back the MsgStr passed in. **Error:** If an error occurs trying to get a message from the resource file, a message will be logged to the error log.

```
class ErrorLog: protected LogManager
{
private:
        LogLevel ErrorLogLevel;
public:
        ErrorLog(ServerId, LogLevel=0, Size=10);
        LogError(ThreadId, ErrorNum, ErrorMsgStr, ...);
};
```

LogError	**Description:** Writes output to error log file. 1. Check that the message level against the current ErrorLogLevel. 2. Format the message and call the long form of **LogMessage** to write the buffer out to the file. **Input:** 1. ThreadId: Thread identifier to help with the debugging process. 2. ErrorNum: Error number used to uniquely identify an error message in the resource file. 3. ErrorMsgStr: Message string which includes stdio like string formatting. 4. ...: variable list of arguments to be inserted into the message string per the format. **Output:** None. **Error:**

Testing design

Each of the components that make up the Server Component Framework will be able to be tested independently of the other components. Each component will have a main entry point defined within a testing .exe to accomplish the Unit testing phase.

Testing of the component framework will be done in phases. Each of the phases is described below along with its dependencies.

Phase 1: Unit testing Test basic components that make up the framework. Each components functionality, restrictions, and boundary conditions will be tested. Will allow testing common configurations for a single server component. This round of unit testing will test the integrated component utilities and framework.	**Dependencies:** 1. ServerComponent class 2. ServerStateMgr class 3. ArgList class 4. Logging Utilities 5. Configuration Manager (flat-file)
Phase 2: Unit testing (full functionality) Test full functionality including messaging interfaces and database connectivity.	**Dependencies:** 1. Phase 1 2. Database connectivity 3. Messaging Service
Phase 3: Integration Testing	**Dependencies:** 1. Phase 2 2. System Monitor (including backup) 3. SLiM Server, App Server, Web-Server
Phase 4: Stress Testing See section on stress testing for details	**Dependencies:** 1. Phase 3

Unit testing plans

Command Line Utilities

The Command line utilities will be tested in a stand-alone module called cmdline.exe. It will support the command line arguments defined in this document.

Configuration Manager

The configuration module is a stand-alone module which will be tested using a config-test.exe executable. The executable will exercise all of the interfaces described above. The configtest.exe executable should be testable in the DB and the non-DB mode.

Logging Utilities

The logging utility will be built as a DLL (otlog.dll). We will provide a binary otlog-test.exe which will exercise each of the interfaces mentioned above.

Server State Manager

The Server State Manager and the Server Component Framework will be tested independently of specialized components. The routines that require specialization (**StartProcessing, StopProcessing, HandleError** and **UpdateConfig**) will be provided to simply return successfully.

Stress testing plans

Stress testing will require having at least the System Monitor functionality implemented since it is used to drive the server components.

1. Test to repeatedly start, stop, reconfigure the server component.
2. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
3. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
4. Test lost database connectivity
5. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
6. Test error recovery under adverse conditions.
7. Test recovery from running out of memory, thread resources.
8. Test recovery from threads dying.
9. etc.

Coverage testing plans

1. Goal: 100% path flow coverage. Only exceptions for known error conditions that cannot be practically reached (e.g. thread synchronization, etc.)

Cross-component testing plans

The following pair-wise testing will be performed:

1. framework/database (phase 2)
2. framework/messaging (phase 2)
3. framework (System Monitor) /framework (backup Monitor) (phase 3)
4. framework/Web Server (phase 3)
5. framework (System Monitor) /framework (Other Servers) (phase 3)

Upgrading/Supportability/Deployment design

1. Each error condition will be documented with explanations and practical work-arounds
2. Component framework will support enhanced debug option to dump additional de-bugging information to special log files.

Open Issues

eStream System Monitor Low Level Design

Michael Beckmann
[redacted]

Functionality

The role of the System Monitor is to monitor the state of the Application Servers and SLiM servers within an eStream deployment. In addition, it also manages a back-up System Monitor.

1. The System Monitor provides the following key services:
    a. Monitors and reports server load across machines
    b. Monitors server state across machines.
    c. Acts as a communication conduit to the database for configuration information needed by the server components.
    d. Initiates state changes within the server.
        i. Start/Stop servers
        ii. Sends requests for servers to update their configurations
2. The system monitor runs as it's own process. Within a multi-system deployment, there will be at least two monitoring processes, each on a different machine:
    a. One monitoring process will act as a primary and the others as backups.
    b. In the event that the primary monitor goes down, one of the backups will take over the primary monitoring responsibilities.
3. The monitoring process can re-launch a server side process if a process terminates unexpectedly (this is a configurable option).
4. The system monitor manages server state by maintaining regular communication via a heart beat protocol between itself, the backup monitors, and every logical server.
5. The system monitor's heart beat supports a light-weight messaging protocol between components to initiate state changes.
    a. Simple request pulse.
    b. Stop request pulse
    c. Configuration request pulse
6. The monitor will raise an alarm if it does not receive a heart beat response from the servers within a specified period of time.
7. The system monitor's heart beat request rate is a dynamically configurable parameter maintained within the database.
    a. The rate can be changed through the administrative UI.

System Monitor Component Overview:

The System Monitor is made up of the following distinct components:

1. Server Component Framework (described in detail in the Server Component Framework Low Level Design)
    a. Server State Manager
    b. Configuration Manager
    c. Messaging Utility
    d. Logging Utility
    e. Command line Utilities
    f. Thread Management Utilities
2. Load Monitor
3. Server Component Manager
    a. Remote Process Launching Utilities
    b. Heart-beat protocol
4. Database Request Manager
    a. Database request protocol

Server Component Framework:

The System Monitor is extended from *the Server Component Framework* by providing an implementation for the following plug-able interfaces: **StartProcessing, StopProcessing, UpdateConfig,** and **HandleError.** Each of the interfaces is described in detail below. The routines **StartServer** and **StopServer**, which are also part of the interface are inherited from the Server Component Framework and do not need to be provided by the System Monitor.

System Monitor State Transitions:

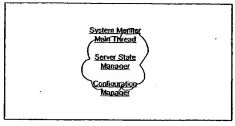
Since the System Monitor is extended from the Server Component Framework, it needs to provide the four interface implementations called out above for each state transition. This section gives a brief overview of the primary state transitions and what is going on within the System Monitor for each transition.

STOPPED -> IDLE:

When the System Monitor is initially started by the WebServer it transitions from the **STOPPED** state to the **IDLE** state via a call to the **StartServer** routine. In this transition the following events occur:

1. Configuration manager reads common configuration from the database.
2. Apply configuration and initialize (System Monitor)
3. Change state to **IDLE** state.
4. Send acknowledgment

In the **IDLE** state the System Monitor is still maintaining only its main thread which it inherits from the Server Component Framework and runs the Server State Manager, Configuration Manager, and the Messaging Service:

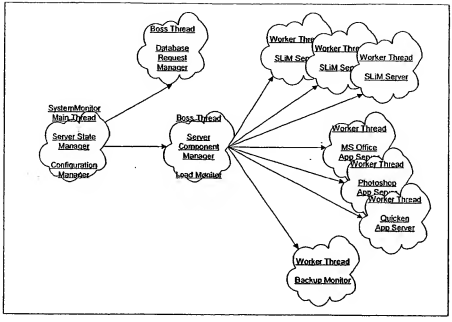


IDLE->PROCESSING

When the System Monitor transitions from the **IDLE** state into the **PROCESSING** state via a call to **StartProcessing** it creates a number of processing threads to do its work. In **PROCESSING** mode the System Monitor employs a "Boss-Worker" parallel programming model:

1. The System Monitor creates two processing threads that acts as the "boss" threads.
2. The first thread is for the *Server Component Manager* to manage each and every server in the deployment. The thread entry point is **MonitorServers**
3. The second boss thread is used for the *Database Request Manager* to service database requests from each of the server components for configuration and any special application server requests. The thread entry point is **ProcessDbRequests**
4. The boss thread *(Server Component Manager)* reads the common configuration for all server components and spawns worker threads for each logical server component. A thread will be allocated to monitor each logical server processes. The thread entry point for each of the worker threads is **ManageProcess**
5. The *Load Monitor* is initialized and runs as a service within the Server Component Manager boss thread.

Below is an illustration of the thread model/architecture when the System Monitor is **PROCESSING** state.



Load Monitor:

The *Load Monitor's* role is to aggregate the load information for each server component within the deployment and update the information to the database.

The *Load Monitor* receives regular load information from each of the server component processes via the *Server Component Manager*. The load information is provided in the acknowledgement to regular heart-beat requests that each of the worker threads initiates with the server process that it is managing.

The *Load Monitor* runs in the "boss" processing thread of the Server Component Manager. The Load Monitor provides thread safe interfaces.

For each application being served within the deployment the *Load Monitor* maintains a list of servers and their response times. It periodically updates the database with these lists which are then consumed by the SLiM servers. The frequency at which it updates the database is configurable.

Server Component Manager:

The System Monitor manages each and every server component process within a deployment. This management activity is undertaken by the *Server Component Manager*.

The *Server Component Manager* spawns a worker thread for each server components within a deployment. The primary entry point for each of the worker threads is **ManageProcess**. This routine has the responsibility of starting and stopping each process and tracking its state, as well as requesting state changes in the component it is managing.

Launching and Terminating Remote Processes:

The System Monitor has the responsibility of launching server component processes across the deployment. The deployment can be made up of a heterogeneous set of servers. In order to accomplish this the system monitor communicates with either a Unix daemon or an NT service that runs on every server within the installation.

The service/daemon is a very simple process that listens on its connection for requests to launch or terminate a process.

If the daemon dies or is accidentally killed, the system will automatically re-launch the daemon. There are facilities for this on Unix and on NT.

The System Monitor communicates with the service via the EMS utilities and a unique protocol defined below in the interfaces section.

Once a process is launched the daemon/service does not track status or trap output of the process (fire and forget). Tracking of the process will be initiated by the System Monitor via the heart-beat protocol.

Database Request Manager:

In order to save database licensing costs, it has been decided that the Application Servers not have direct database connectivity. Instead, the Application Servers will depend on the System Monitor as an intermediary to read and write data from the system database.

The System Monitor is used on behalf of all the other servers to interface with the database for retrieving configuration information and/or any other information that the Application servers may need.

The Database Request Manager is launched from the **StartProcessing** interface. It is launched in its own boss thread. The thread entry point is **ProcessDbRequests**.

The Database Request Manager also defines a protocol for each of the database interfaces defined in the WebServer/Database interface document.

Data type definitions

The System Monitor maintains a few key data structures beyond what is provided by the *Server Component Framework*:

Request Queue:

The System Monitor makes use of mutex protected queues to communicate between threads. For example, each worker thread will maintain its own mutex protected input queue. The following components will maintain input queues:

1. Boss thread for Server Component Manager
2. Boss thread for Database Request Manager
3. Each server monitor worker thread

A queue entry models, fairly closely, the data contained within the messaging protocol between servers as defined in the *Server Component Framework*. This data can be used for the input queues for each worker thread managing individual work components.

The details of the input queues themselves is defined in the common thread API.

```
struct QueueEntry {
        int Request
        ServerID SenderID
        ServerID RecieverID
        ServerID TargetID
        Data
}
```

Managing Worker Threads:

The System Monitor's boss thread (**MonitorServers**) maintains an in memory list of all known servers within a deployment, and their associated worker thread ID. In addition, it maintains information about request queues for each worker thread. This list is maintained as the **ServerInfoList**. The list is made up of entries which maintains all the necessary information to start/stop/manage/communicate/etc. with a server component. The **ServerInfoList** is traversed by the Server Component Manager and the Load Monitor.

```
struct ServerInfoEntry
{
        ServerConfig* Config;       //configuration  for each server component
        Thread ThreadID;            //Thread ID in which the Server Component
                                    //Manager is executing
        ThreadQueue* InputQueue; // processing queue for this thread (defined in thread
                                    // utilities package.
};

typedef vector <ServerInfoEntry> ServerInfoList;
```

Load Monitor:

The Load Monitor maintains the following data structures:

```
struct ServerLoadEntry
{
        ServerID
        LoadInfo  // still needs to be defined
};
struct ServerSetEntry
{
        int AppID
        list <ServerLoadEntry> ServerSet // ordered list of application servers
};
typedef vector <ServerSetEntry> LoadMonitorList;

class LoadMonitor            // thread safe/ re-entrant
{
private:
        LoadMonitorList List;
public:
        LoadMonitor();
        ~LoadMonitor();
        UpdateLoad(ServerID, LoadInfo);
};
```

Server Monitor:

```
int MonitorServers(void);  // non-member function used as thread entry point
int ManageProcess(vector <ServerInfoEntry>); // non-member function used as entry
                                                // to for worker threads
```

System Monitor Configurations:

Configurations unique to System Monitors include:

Information	Supports Dynamic Config	State	Description
LoadUpdateRate	Yes	All	Frequency at which the Load Monitor will update the database with aggregate load information. Value is specified in milliseconds. If the value is zero then the Load Monitor will not update the database essentially disabling load monitoring.

Database Interface Protocol:
There will be a unique operation code for each database interface. Likewise there will be an acknowledgement.

OpCode	senderID	receiverID	targetID	Data

Interface definitions

Server Component Framework:

The implementation of the System Monitor is derived from the generic *ServerComponent* class. Refer to the *Server Component Framework* Low Level Design.

Implementing a *SeverComponent* requires the definition of the following methods:

StartServer	**Description:** This routine is called to start a server process and bring it to the **IDLE** state. It assumes that the server is in the **STOPPED** state. This routine performs all the necessary initialization and configuration common to all server components. This function is predefined in the *ServerComponent* base class. The System Monitor has no need to override it. **Input:** None. **Output:** Integer value designating return status. Success = 0. **Errors:**

StopServer	**Description:** The routine StopServer is called to cleanup and terminate a server process. It assumes that the server component is in its **IDLE** state and is therefore not processing any requests. This function is predefined in the *ServerComponent* base class. The System Monitor has no need to override it. **Input:** None. **Output:** Integer value designating return status. Success = 0. **Errors:**

StartProcessing	**Description:** This routine initiates all the activities unique to the System Monitor: 1. Launches a boss thread. Call **MonitorServers** as the boss thread entry point. 2. Launch another boss thread. Call **ProcessDbRequests** as the primary entry point. 3. Return immediately Note: The **StartProcessing** routine must return immediately else the Server State Manager will be blocked. **Input:** None. **Output:** Integer value designating return status. Zero if success else error **Errors:** Error launching thread.

StopProcessing	**Description:** This routine will perform the following activities: 1. Terminates each server component's monitoring thread. This effectively disables the monitor from managing any executing server components including stopping existing components or starting new ones or servicing configuration requests. 2. Terminates the Database Request Manager. Note: The primary system monitor will only execute this interface if a system administrator explicitly changes the monitors state or an error has occurred within the monitor. **Input:** None. **Output:** Integer value designating return status. **Errors:**

UpdateConfig	**Description:** This routine will perform configuration changes specific to system monitor functionality. Refer to the section on system monitor configurations. This routine can be executed from both the IDLE and PROCESSING states. **Input:** None. **Output:** Integer value designating success or failure. Zero for success. **Errors:**

HandleError	**Description:** This routine is called by the Server State Manager whenever an error is encountered. If this routine can handle the error then the original calling state will be maintained. If the error cannot be handled the process will transition to an ERROR state and terminate. **Input:** None **Output:** Integer value designating success or failure. zero for success. **Errors:** Error specifying that the error could not be handled.

Server Monitor:

MonitorServers	**Description:** This routine is defined as a non-member function and is the thread entry point for the "boss" thread of the Server Component Manager. It is the primary thread for managing and monitoring all the server components. 1. Create an input request queue for this thread 2. Start Load Monitor 3. Go to the database and get a list of all the server components calling **GetServers** as defined in the Database Low Level design document. 4. For each server component retrieve its common configurations calling **GetServerConfig** and create and entry into the **ServerInfoList** 5. Spawn a worker thread to manage/monitor the backup System Monitor. Call **ManageProcess** as the thread entry point passing in the **ServerInfoEntry** After the backup monitor has been launched, repeat step 4 for each server component. 6. Check the input queue for new worker requests. 7. Loop back and redo steps 3-6 looking for new servers to manage **Input:** None. **Output:** **Errors:**

ManageProcess	**Description:** Non-member function that launches and monitors the specified server component processes. It is expected to be the start routine for its own thread. This routine is called from **MonitorServers.** Performs the following steps: 1. Call **StartRemoteProcess** to launch the server component per the configuration information provided in the **ServerInfoEntry**. 2. Open a point-to-point messaging connection to the newly launched server component. 3. Initiate a regular heart-beat request at the rate defined within the configuration. 4. Enter monitoring loop a. Check for requests from the input queue for this server component. b. Send the request to the server currently being monitored by this thread. c. Listen and wait for response; timeout if wait is too long. d. If a response is received update state information in **ServerInfoEntry** and update the database as necessary calling **SetServerState** and **SetServerLog**. e. Enqueue load information calling **UpdateLoad** f. repeat a-f **Input: ServerInfoEntry** **Output:** Status of the server component on exit. **Errors:** 1. launch error 2. messaging error 3. unexpected server component termination

Database Request Manager:

ProcessDbRequests	**Description:** This routine is the thread entry point for the Database Request Manager. It is called from **StartProcessing**. 1. Connect to Database 2. Create a listener using EMS utility 3. Sleep until a request comes in and then enqueue the request. 4. Worker threads will then dequeue and satisfy the db requests and terminate **Input:** db connectivity info **Output:** **Errors:**

Load Monitor:

The *Load Monitor* maintains three public interfaces:

UpdateLoad	**Description:** This routine updates, in memory, server sets with the current load information. This routine is thread safe/re-entrant This routine is called by **ManageProcess** 1. Traverse server lists by application. If the application. If the application entry is not found create it and add it to the list. 2. If the application entry is found search for the server entry 3. Update load information 4. Call **FlushLoadData** per the database access frequency configuration. **Input:** Load information, ServerId **Output:** Integer value specifying Success or Failure. Zero if success. **Errors:** Error processing load information
LoadMonitor	**Description:** Initialization routine for the Load Monitor. Called by **MonitorServers** 1. Initialize server lists to NULL. **Input:** None. **Output:** None. **Errors:**
~LoadMonitor	**Description:** Destructor for the LoadMonitor. 1. For each server flush all load data to the database calling **FlushLoadData** 2. Cleanup server set data structures and exit. **Input:** None. **Output:** None **Errors:**

Primary and Backup System Monitor:

The backup System Monitor needs to be able to take over the primary system monitor responsibilities in the event that the backup loses communication with the primary. The following interface will cause the backup monitor to take over as primary.

SwitchPrimaryMonitor	**Description:** Take over primary monitor responsibilities. This routine is called from the **HandleError** routine defined for a backup monitor. If the backup monitor doesn't hear from the primary monitor, it assumes that the primary monitor has died. 1. Update database validating monitor switch. 2. If no database connectivity then go back to listening mode until the next time out. 3. If database connectivity then attempt to shut down the old primary System Monitor just in case it is still running. 4. Go through the same steps as the primary monitor would do when its **StartProcessing** routine is called including starting up a new backup system monitor **Input:** None **Output:** integer value designating success or failure **Error:**

Launching and Terminating Server Processes:

TerminateRemoteProcess	**Description:** This routine terminates a remote process either on a local machine or on a remote machine. This routine is called by **ManageProcess**. 1. Verifies that the target machine is up and running 2. Send message to daemon/service to terminate the process. Uses EMS messaging service. **Input:** Process info, target machine **Output:** integer return specifying success or failure. **Errors:**

StartRemoteProcess	**Description:** This routine spawns the requested process either on the local machine or on a remote machine. This routine is called by **ManageProcess** 1. Verifies that the target machine is up and running by sending ping to daemon/service on the specified machine. 2. Send message to daemon/service to launch a process passing the process path and any arguments. **Input:** Process name, target machine, attributes **Output:** integer return where 0 designates success else error code is returned. If successful the routine will also return process info **Errors:** 1. machine not responding 2. executable not found 3. failure on launch

Daemon/Service	**Description:** Unix daemon/NT service which runs on every server machine within the deployment. Its role is to dispatch requests to create and terminate processes, obtain process information. 1. Initialize/Create a listener using EMS 2. Go to sleep until a request comes in 3. Dispatch request 4. Return information to requestor 5. Perform any necessary logging 6. Repeat 2-6 **Input:** listening port **Output:** None **Errors:** Errors will be written to a special log file

Testing design

The System Monitor will provide a command line option to facilitate testing the component in its various testing phases. The option will allow the system to manage its interfaces through flat files.

Unit testing plans

To unit test the System Monitor it will require a single generic server component running on the same machine.

Phase	Description	Dependencies
Phase I: Local process management	Cycle through all the state changes for a single local generic server component	1. Functionally complete Common Server Framework 2. Generic Server component
Phase II: Remote process management	Same as Phase I for remote processes	1. Phase 1 dependencies 2. Server daemons
Phase III: backup monitor failover	Verify that the backup monitor fails over and continues to manage the system	1. Phase II dependencies 2. Backup Monitor Server

Stress testing plans

Tests should include:

1. Max # of generic server components on a single machine.
2. Max # of generic components across multiple machines.
3. Max # of generic servers changing state at least once per second
4. Test to repeatedly start, stop, reconfigure each server component.
5. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. Auto Restart.
6. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
    c. Auto Restart.
7. Test lost database connectivity
8. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
9. Test error recovery under adverse conditions.
10. Test recovery from running out of memory, thread resources.

11. Test recovery from threads dying.

Coverage testing plans

The System Monitor will achieve 100% code coverage with the exception of error conditions which are possible however difficult to reach in practice.

Cross-component testing plans

The System Monitor interacts with the following components:

1. System Monitor/Database
2. System Monitor/WebServer
3. System Monitor/Server Component Framework (other servers)
4. System Monitor/Backup Monitor

Upgrading/Supportability/Deployment design

1. All diagnostics will be documented as to their root cause and workarounds/actions to be taken.
2. The system monitor will support an enhanced debug support which dumps additional information to special debug logs.

Open Issues

1. Need to identify a mechanism to ensure that we do not have more than one primary monitor running at any given time.

eStream Messaging Service Low Level Design

Version 1.1

Sameer Panwar
[redacted]

Functionality

Estream is fundamentally a networked system, and thus must rely on a communication infrastructure as part of its foundation. The EMS subsystem provides the API for eStream processes to communicate with remote eStream processes. It is subdivided into layers and is intended to freely support different combinations of choices from each layer in an extensible manner. E.g. XDR over HTTP over SSL, while new layers can be added in the future (for compression, custom encryption, etc.). This is important considering it will be used across the variety of eStream components, including the client cache manager and all the servers.

The layers are enumerated as follows:

1. Packaging Utility [XDR/SOAP/name-value pairs, etc.]
2. Procedure Call Layer (PCL)
3. User Layer(s)
4. Message Transport Layer (MTL) [SSL+TCP/HTTP+TCP/MSG+TCP]

The Packaging Utility is primarily used to pack and unpack the parameters of the message into the body. It isn't really a true layer, in that it is visible to the other layers—they also use it to pack their data into their headers (except if the layer is a given standard). Thus all layers are subject to the same packaging scheme, unless they provide their own internally. The various packaging utilities won't be tied to EMS directly—they will be available for packaging data into any buffer.

The Procedure Call layer provides the functionality to assign API and Function identifiers in order to define a remote API, as well as providing a dispatch mechanism on the server side to enable appropriate processing of incoming messages. This layer will also handle callbacks for replies to asynchronous messages. The implementation for eStream 1.0 for this layer will be called EPC for eStream Procedure Call.

The User layer is present to allow the user of EMS to add custom headers on the send side and perform early checks (e.g. on AccessTokens) on the receive side before being queued for service in the thread pool. This could also be used for compression or a custom security layer, and multiple User layers could be put in place.

The Message Transport Layer is special in that it is this layer that uses the actual socket interface. However, it has the additional functionality (referred to as MSG above) to recognize message boundaries via a magic number and a message size (which is also done

via an HTTP header or an SSL header). This enables us to grab a complete message over the stream-oriented TCP before sending it up to higher layers. Note that UDP might not be supportable in EMS—there may be inherent assumptions that the underlying network protocol is reliable and connection oriented, but I won't know for sure until EMS is implemented for TCP whether it will be adaptable to UDP. Also, the MTL is also responsible for contacting proxy servers in the manner appropriate to the protocol they're using (SSL or HTTP).

Messages are linked lists of memory blocks (which also track size used), each block being a layer header, with the original message body last. They are always flattened in MTL before sending, but can be flattened elsewhere if desired. Probably we'll use 4k blocks for each header, and a 64k block for the body. This linked list of blocks is called the Message Buffer, which is contained in the Message object. Messages also contain per-message info relevant to the various layers, including API & FUNC #'s, Message IDs, etc. This per-message info is used to generate the appropriate headers for outgoing messages, and is unpacked into the Message for incoming messages. It is the sum of all per-message info any layer might want, placed in the single Message class, even though obviously only the layers in use for the given connection will actually be active.

EMS layers have:

Header data members
BuildHeader function
BodyXform transformation function (e.g. compression, but often just null)
Send function
Internal Connection info (nonces, request ids, SSL_CTX, etc.)
Reset function (when need to re-establish a connection)
Session info that's global for the layer, static data (session ids, callbacks, etc.)

In usage, each layer, in the outgoing direction

1. Gets outgoing Message, becomes owner of the memory.
2. Builds a header (needs to grab a free block) using per-message info set up by the user in the Message, and adds it to the Message Buffer.
3. Applies any applicable transformation to the message body.
4. Passes the Message it to the lower layer.
5. Each layer is NOT to remember anything on a per message basis, since it won't be notified if the message is lost!

Each layer, in the incoming direction

1. Gets the incoming Message.
2. Unpacks the header, performs any relevant checks and transformations on the message body (that is, the body relative to this layer's header). Increments a pointer so that a higher layer can know where its header starts.
3. Places any relevant header info into the per-message info in the Message, e.g. Message IDs. This layer-related info will thus be available outside EMS. Only a limited number of these will actually get set, since some of these are only relevant in the outgoing direction.

4. Passes the Message (or an error) to the higher layer. If the higher layer is the EMS caller, then the Message is delivered to them so that they can then unpack their data.

Steps to use EMS:

1. First, a Connection object must be initialized, describing the remote server, the stack of layers and packaging utility, along with other configuration.
2. Next, the user must package their data into a Message Buffer.
3. Then they must set up any required per-message info in the Message object.
4. Finally, they can send the Message (this triggers the actual layer processing, and then the Message gets queued to go out the network).

Message Stack

A Message Stack is an object that describes the order of the actual layers being used for a given remote server, and also binds to a packaging utility, which is used by the layers and user to marshal and unmarshal data. The Message Stack doesn't actually contain the layer objects themselves, just pointers, so there is just one Message Stack object. For any listening port on a server, only one Message Stack is used, and essentially defines the language spoken on that port. Other ports on the same server can use different Message Stacks.

Connection

The Connection object contains all relevant data EMS requires to manage a connection with a remove server. It contains the relevant layer objects, which keep track of connection-level info, as well as a Message Stack. It also contains the remote server's DNS name and/or IP address and port number, and the same for a proxy server if necessary, these being used by the Message Transfer Layer. The reply timeout and socket handle and state are also maintained here. There is one Connection object for every combination of layers (and packaging utility) and remote server we care to use, and instantiates each layer object and initializes it (and assembles the Message Stack) as part of its constructor. A given connection will be configured to support synchronous or asynchronous communications, but not both. A Connection also contains an outgoing message queue, and a (pointer to a) buffer for messages in the process of being read or written to the network along with a current position index.

Connection Management

I/O can be nonblocking, but the first 3 layers **are not allowed to block on an OS call**, so stuff only gets queued in the MTL. Thus there is one well defined place where sends/receives of messages can be resumed, and the states involved are described below. If the socket is blocking, then we need to set a timeout and return an error on a failure.

A special thread called the socket watcher does a select on active sockets, and triggers actions based on the socket's connection state and message state. If the state isn't CONNECTED, then it continues the relevant handshake. If it is CONNECTED, then it sends

the outgoing message, or dispatches an incoming one. It is also responsible for handling timeouts.

Connection states:

DISCONNECTED	
TCP_HANDSHAKE_WAIT	select(write/except)
MTL_HANDSHAKE_R_WAIT	select(read)
MTL_HANDSHAKE_W_WAIT	select(write)
CONNECTED	select(read)

Message state (outgoing) for each connection:

EMPTY	select(read)
MSG_READY	select(read/write)
MSG_SENDING	select(read/write)

Relationships:
1 Connection Object per remote server, and 1 associated Message Stack for the Connection Object. Thus, if communication to the same server is required using two different Message Stacks, then 2 Connection Objects are required. Multiple Connection Objects with the same Message Stack to the same server are allowed for clients, but discouraged, since that eats up server resources.

Handshakes

Some of the underlying protocols need to perform handshakes before they can exchange user data. E.g. SSL needs to perform authentication and key exchange and obtain a session before the user's data can be encrypted for transmission. These handshakes can take more than one exchange to establish the session, e.g. SSL takes 2 ½ round trips (on top of TCP's 1 ½ round trips). SSL's interface for handshaking supports nonblocking I/O by implementing an internal state machine to handle the handshake protocol. EMS will export this interface on the send side (by keeping a connection state); for receives, EMS will have to check to see what the connection is waiting on (read or write, even if the user request was different) and check for it during a select(). If data is returned when the connection is still being set up, then the handshake is continued to read the waiting data. After the handshake is complete, the layer must check for any pending message, process it with the session data, and pass it down. While a handshake is going on, all new requests are processed down to the MTL and then queued. This means that header data members relating to the MTL cannot be changed, so access to them will be through functions which will fail on a call for a given connection object that is not disconnected. Handshakes are NOT supported at any higher layer than the MTL, since that would entail too much complexity. Thus any other desired handshaking must be implemented on top of EMS, though handshaking could be possibly supported within higher layers if communication were only synchronous, if the need arises. A successful SSL handshake will set the session ID. If the handshake fails on trying to re-use an old session ID that expired, then the session ID is cleared, and a full handshake is required again.

Global EMS Stuff

EMS at the global level manages the messages being sent. When timeouts are being used, EMS can assume that for every outgoing message some response message must be returned from the server, and will report a timeout to the caller if a response is not received by the deadline. EMS requires a Message ID (also used by the PCL dispatcher) to be passed in by the user, which it stamps on the outgoing message, and is present on the reply message which cancels the timeout. Message IDs are also used by the PCL dispatcher, so that the client callback can match the response with its request. EMS requires that the user use a monotonically increasing Message ID, and will thus reject any reuse (this enforces uniqueness of Message IDs for the lifetime of the process). Since Message IDs are low-level EMS details, they live in the MTL. EMS keeps an ordered queue of Message IDs and timeouts (but throws away the actual message contents once it's sent), and the socket watcher thread uses the earliest timeout in its select() call.

Asynchronous calls

For asynchronous calls, client callbacks are registered one for each remote function (these are managed by the PCL), as well as a timeout function (which is handled by the MTL). The socket watcher thread gets messages as they come in, and the PCL will queue them for a dispatcher thread to process.

Buffer/Memory Management

Since we need to manage buffers efficiently, and also avoid possible attacks if we support very large message sizes, we should pick some reasonable maximum size for any message. For now, I will set that to be 64k (remember, this is the max size after packaging), which really may only affect the app server. This maximum size, of course, includes headers in any of our layers, and may be restricted even more when SSL is in use (due to its own restrictions). We should also consider the lifetime of memory buffers. In time sequence, normally what happens is that the user grabs a buffer for the main message body, packs their data into it, then passes it to the top EMS layer. Each EMS layer then grabs a buffer for the header, and at the last step this is flattened into yet another buffer (which frees the previous buffers), which is then passed to the network, after which the Message object can be freed. When a message is received, it is placed into a buffer in a Message object, which is given to the user, who is then responsible for freeing the Message object when they're done with it. Before that, the unpackaging functions may allocate memory for variable sized objects, which then also become the user's responsibility to free.

Since the lifetime of many of the memory buffers is short, but the number of outstanding messages is generally pretty low, reuse of buffers can definitely help efficiency. For the various transient classes, I expect to implement a memory manager class that will keep the objects around instead of freeing the memory, and then reuse them when a new one is needed. Thus each object will have its own pool. Buffers will be managed the same way, probably coming in 4k (for headers) and 64k sizes. However, I'm not sure if I want to use this approach for memory allocated for the variable sized objects in the unpackaging functions; maybe I'll just use malloc for that.

Socket Watcher

The socket watcher thread is required to support asynchronous messaging, by grabbing and dispatching messages as they come in, also making it the logical place to handle timeouts for asynchronous threads. This special thread needs two data structures—the list of sockets to watch and a queue of timeouts. Connections that are configured as synchronous never pass their sockets to the socket watcher. For asynchronous connections, whenever the connection state changes, the socket may be added/updated in the socket watcher's list (we need to know whether to check if the socket can be read from or maybe written to, depending on the connection state). When the MTL discovers that the TCP connection was closed, it removes it from the socket watcher's list. Note that the socket watcher is actually "just below" the MTL: it doesn't know how to decode a Message ID. When a socket is ready for reading, it gets the associated Connection object, and then calls the MTL (in the socket watcher's context) to pull the message out and then cancel the pending timeout. Then, still in the socket watcher thread, the message is pushed up the Message Stack until it is dispatched to another thread for actual servicing (as in the PCL). Then the socket watcher goes on to process any other ready sockets. Note that thus the socket watcher could process messages going to different MTL's. There is only a single socket watcher thread in EMS for a given process. The socket watcher also checks for incoming connections on the listening ports (there can be more than one on a server).

The socket watcher is also responsible for completing sends of messages that were left incomplete (i.e. would have blocked) or are queued. A connection's message state reflects this and has the watcher check for socket writability—when that happens, it then tries to complete the send as appropriate. In fact, if asynchronous messaging is being used, all actual writes over the network happen in the socket watcher thread, the user threads just dumping the messages into the queue and changing the connection state appropriately.

When a user uses synchronous messaging, this is different in that the user's own thread then sends the outgoing message, waiting until the send is complete, and then does a select on just that one socket with its own timeout. When the reply is received, then the MTL and higher layers are executed in the user's context, and no dispatch to another thread occurs (since the Connection is configured as synchronous).

Clients vs. Servers

Client and Server behavior is distinguished by the EMS calls they use. Clients use EMSMsgSend(…) which takes a Message ID and a timeout value, and if they're making asynch calls, then they must also set up callbacks. Servers use EMSMsgReply(…) which returns the above Message ID and has no timeout. Despite this difference, the socket watcher behaves the same on the client and the server. It simply does a select on all open connections and tracks timeouts. Of course, the server just disables the timeouts. Also, the server listens on a specific socket for incoming connections, while the client does not. However, servers have a different kind of timeout—the connection idle timeout, which means if a connection isn't used by a client for a certain interval, the server will drop the connection to conserve resources (sockets). The server's MTL will establish its own

callback with the socket watcher in order to close the connection when it times out (and it will also cancel timeouts whenever it receives a complete Message). The client's MTL will transparently detect the dropped connection and reestablish it when the next outgoing message is queued.

Another difference on the receive side occurs in the PCL. There, the client side PCL will look at the PCL function number and dispatch to a callback (if the call was asynchronous), or pass it up to the user directly (since the watcher thread wasn't involved). On the server, only the socket watcher will get messages, and the PCL will dispatch the messages to the functions that were registered for the corresponding PCL function number. Hmm.. These are actually similar enough that maybe they could be done with the same mechanism. Then the main difference between the client and server with respect to EMS is just that the server is listening for incoming connections and uses different timeouts.

Errors

I can't really describe this in detail now. This will have to come out of the implementation.

In general, if an incoming message is bad in some way, it'll just get dropped. If we care, we can keep a counter that is incremented for each bad message and raise an alarm if it reaches a threshold...

Multithreading/Synchronization

EMS will have 2 threads internal to itself, both of which are only required when asynchronous messages are configured or EMS is being used in a server process. The first thread is the socket watcher, which is started by the EMS initialization. The second thread is the PCL dispatcher, which is started by the PCL initialization (this thread is the context in which actual requests are processed and work is done, and this could in fact be a pool of threads with a queue in front). Other layers could add their own threads if desired. All other EMS functions are fully executed in the context of the caller.
What are the shared data structures among threads? Since EMS functions could be called by different non-EMS threads, there's a good number of data structures that need to be synchronized.

First, since multiple messages may be processed at the same time, the layers above the MTL need to lock any shared data that they manipulate internally (such as IDs or nonces). Also, access to the memory buffer pools must be synchronized. Once the MTL is reached and the message is ready to be sent through the MTL, a lock is acquired (e.g. to protect the SSL_CTX structure) and only released when the bottom-level SSL or TCP interface returns, so only one thread can be blocked on a recv for a given socket at any time. The Connection's message queue and connection state must also be synchronized, as well as the socket watcher's timeout queue and active socket list. For connection-level stuff, it may be easier to lock the entire connection object (i.e. the whole time from the message body being passed to the top layer and the point where the assembled message is placed on the outgoing queue, thus covering all layers at once); this will probably be implemented first, and we can use finer grained locking if this is a performance issue.

General policies

EMS does not ensure any ordering of processing of requests. Requests of course go out in the order they were sent, but the server may process them in a different order due to thread concurrency. If ordering is desired, it needs to be implemented in a User layer via some kind of queuing mechanism.

HTTP

[mostly taken from Bhaven's HTTP document]

To make HTTP proxies happy, we'll implement a simpleminded subset of HTTP headers. For the request message, we'll use the format:

```
POST / HTTP1.1
Host: <servername>
Connection: KeepAlive
Content-Type: octet-stream
Content-length: <content_length>

<message body>
```

For the reply message, we'll use the format:

```
HTTP/1.1 200 OK
Content-Type: octet-stream
Content-Length: <content_length>

<message body>
```

The HTTP+TCP MTL in EMS will also support a GET-type request with a URL but no message body, if the appropriate Message options are selected and the user provides the URL string, which they (or some higher layer) will have to package. The server-side HTTP+TCP MTL won't decode the URL, it'll just store it in the Message, and leave it to higher layers or the user to interpret it. This layer will assume the HTTP header will have a max size of 4kB, and report an error if it pulls more than out of the socket before reaching the body.

Also, this layer may need to emulate HTTP 1.0 behavior (i.e. no persistent connections) if the proxy isn't 1.1-friendly, but I'll leave that to be implemented later if necessary, and for now we'll notify our customers that they need a 1.1-compliant proxy server. From what we've gathered so far, most deployed proxies should be 1.1-friendly.

EStream 1.0

The current plan for eStream 1.0 is to implement the following layers:

Packaging Utility – adapt the Sun XDR routines. We'll add more for the client←→Web server interface or Web server←→SLiM interface later if required.
Procedure Call Layer – we'll just implement one, since only one is required and all communications we have planned will use it.

User Layer – unknown, but we could implement AccessToken checks at this layer, before requests are queued at dispatch time.
MTL – we will support SSL on TCP and HTTP on TCP.

Synchronous messaging could be implemented first, but pretty much all of the asynchronous messaging needs to be in place for server functionality. Therefore release 1.0 will have full asynchronous messaging support.

Data structures/Interfaces

First things first. EMS needs to be initialized on process startup.

EMSInitialize(???) – this function sets up any global stuff required.

```
class EMSBlock
{
        char * buf;
        uint32 bufsize;  // size of the alloc'd mem that buf points to, for bounds checking
        uint32 size;     // amount of buf that is occupied, i.e. pointer to where to add data.
}

class EMSMessage
{
public:
        // actual message contents
        List<EMSBlock>      MessageBuf;

        // PCL stuff for this message
        uint32 apiNum;
        uint32 funcNum;

        // MTL stuff for this message
        uint32 messageId
        (SSL related stuff perhaps)

        // HTTP stuff for this message
        boolean isRequest;
        char * URLString;

        // other stuff
        uint32 position;  // index into MessageBuf where next layer should start reading
                          // or writing. Also used to indicate where more data should be
                          // read or written to from the network if the Msg is incomplete.
        flatten(); // this function flattens MessageBuf into just one block
}
```

```
class EMSConnectionBase
{
public:
        (functions to set some of the below values)
        SendMsg(EMSMessage * msg);  // passes msg through the stack
        EMSMessage * GetMsg();
        Reset(); /* disconnects, resets all state (clears Q's), can then be reused for other
                * servers with the same MessageStack as the given Connection object */
private:
        char * destName;
        addr   destIpAddr;
        uint16 port;
        char * proxyName;
        addr   proxyIpAddr;

        boolean isClient;       // tells EMS how to behave, esp. wrt the timeout
        boolean synchronous; // actually only relevant if is a client
        uint32  timeout;        // in milliseconds
        EMSMessageStack    stack;
        EMSSocket   * sockPtr;
        EMSConnState       state;
        Queue<EMSMessagePtr>     outgoingQ;
        EMSMsgState        msgState;
        EMSMessage *       sendingMessage;
        EMSMessage *       receivingMessage;
}

class EMSConnection_XDR_EPC_SSLTCP : EMSConnectionBase
{
private:
        EMSPackagerXDR     xdr;
        EMSLayerEPC        epc; /* for estream procedure call ? */
        EMSLayerSSLTCP     ssltcp;

        EMSConnection_XDR_EPC_SSLTCP (init stuff for XDR, EPC, SSLTCP layers
as well as dest'n name & IP address, port, timeout)
        {
                the constructor grabs the params to initialize the relevant layers, and
                places the layers in the proper order in the message stack.
        }
}

class EMSConnectionMgr
{
        [This class is used to get connection objects of a given message stack,
        by the socket watcher; actually must pass derived classes in as this type,
```

```
        i.e. the socket watcher wants a "EMSConnectionMgr", but we give it a
        EMSConnectionMgr_XDR_EPC_SSLTCP, and the right stuff happens
        via the virtual functions.]
}

class EMSPackagerBase   [just an abstract base class]
{
public:
        PutUint32(EMSBlock *, uint32) = 0;
        PutUint8(EMSBlock *, uint8) = 0;
        PutString(EMSBlock *, char *) = 0;
        [Etc., one function for every basic datatype supported.
         For custom datatype/structs, should build a routine
         for the struct that takes a packager object as an arg
         and uses its basic datatype packaging routines.]
}

class EMSPackagerXDR
{
        [actually implements the above virtual functions using the XDR spec]
}
[for outside EMS, the XDR functions will be visible as, e.g., EMSPutUint32XDR(char *
buf, int * pos, uint32), where pos points to the place in the buf to place the uint32 and it
gets incremented as appropriate, but these functions won't check bounds for buf.]

class EMSLayerBase   [another abstract base class]
{
public:
        ProcessMsgIn(EMSMessage *) = 0;  [processes an incoming message]
        ProcessMsgOut(EMSMessage *) = 0;  [processes an outgoing message]
        Reset();

private:
        int BuildHeader() = 0;
        int BodyXform() = 0;
}

class EMSLayerEPC : EMSLayerBase
{
public:
        EMSLayerEPC()
        {
                [starts dispatcher thread]
        }
        GlobalRegisterAPI(API #, # funcs, func table);
```

```
private:
        [table of pointers to function callback tables (one table per API),
         this table being statically allocated would mean that I'd set a max
         supported API number to be something like 256.  This table is
        static data, i.e. visible and shared by all instantiations of this class.]
}

class EMSLayerSSLTCP
{
public:
        EMSLayerSSLTCP(SSL params, TCP params)
        { ... }
private:
        uint32 curMessageID;
        SSL_CTX * context;
}

class EMSMessageStack
{
        List<EMSLayer *> layerlist;
        EMSPackager * packer;
}
[this is really just used internally by the Connection object, of little interest anywhere
else]

EMSTimeout
{
        time    deadline;
        uint32  messageID;
        EMSConnection * conn;
}

EMSSocket
{
public:
        ChangeState(EMSSocketState newState)
        [this modifies the FD_SETs below, as do the constructor & destructor.]

private:
        socket  sock;
        EMSSocketState      state;  // derived from connection state & message state
        EMSConnection *             // for other open sockets
        EMSConnectionMgr *          // for listen sockets

        static FD_SET readset;      // used by select()
        static FD_SET writeset;
```

```
        static FD_SET exceptset;
}

EMSRegisterStack(EMSConnectionMgr *, uint16 listen_port)
{
        [tells socket watcher to bind this port with the given connection queue.
        Grabs a free EMSConnection for every connection made with a client
        and associates it with the new socket]
}
```

Most of these interfaces appear because of EMS's layered structure, to support extensibility, but are not meant to be used by the EMS user, and involve lots of lower-level grungy details that are transparent above EMS. In general those coding to use EMS only need to worry about the following:

First, configure global data of layers being used, e.g.

```
EMSLayerEPC::GlobalRegisterAPI( ... )

EMSLayerSSLTCP::DoCertificateStuff( ... )
```

Then, build a connection object appropriate to the server & API you want to use, e.g.

```
foo = new EMSConnection_XDR_EPC_SSL_TCP("appserver.foo.com", &address,
4567, 1000 [timeout in ms], CLIENT, SYNCHRONOUS);
```

Then create a message object and start dumping your parameters into it:

```
msg = new EMSMessage;
msg->apiNum = EPC_API_CLIENT_APP_SERVER;
msg->funcNum = EPC_FUNC_GET_PAGE;
msg->messageId = msgid++;

EMSPutAccessToken(foo, msg, &accessToken)
EMSPutUint32(foo, msg, appId);
/* this is really an inline function that does foo->stack->packer->PutUint32(msg, val) */
EMSPutUint32(foo, msg, fileId);
EMSPutUint32(foo, msg, pageNumber);

foo->SendMsg(msg); /* msg is destroyed now! (maybe should sent &msg?) */
reply = foo->GetMsg();        /* in the synchronous case; this blocks w/ timeout*/
size = EMSGetUint32(foo, reply);
pagedata = EMSGetByteStream(foo, reply); /* this guy allocs mem for you */
```

For reuse, the above can be wrapped in its own function with the signature:

GetPageFromAppServer(foo, &accessToken, appId, fileId, pageNumber, &size, &pagedata);
When you send your next message you should reuse 'foo'. If the server had dropped the connection, it is automatically reestablished. When you're done with foo, you can Reset() it, and reinitialize it for another server, or free it.

That's generally how messages are sent, on the client or server. The server has a few extra things it needs to do at startup time:

xdr_ssl_tcp_mgr = new EMSConnectionMgr_XDR_EPC_SSLTCP;
EMSRegisterStack(xdr_epc_ssltcp_mgr, 4567);

This starts the socket watcher thread (if it doesn't already exist), and tells it to listen to socket 4567 and when a connection is made, it sets up a EMSConnection_XDR_EPC_SSLTCP to pull data off the network when it arrives. Requests coming in are handled by the appropriate layers, which were initialized globally above, so EPC will dispatch the incoming message as configured by EMSLayerEPC::GlobalRegisterAPI(...). The socket watcher will free these connections when the idle timeout expires. The only problem with this approach is that if the EMSConnection needs initialization info that is layer-specific (can't think of any right now that aren't global), there could be problems. The Connections should have default initializations that make sense so they do the right thing when messages come in (or maybe they need to have a special server mode?).

When the client uses asynchronous messaging, it must have done a EMSLayerEPC::GlobalRegisterAPI(...) as well to register its callbacks. When the first asynchronous EMSConnection is created, the socket watcher is started (if not already there). When the connection is actually established with the server, the socket watcher is informed to watch it for the reply (along with the timeout). Otherwise, there is nothing else the client needs to do (other than track message IDs to match replies and timeouts).

Testing design

Unit testing plans

The XDR packaging utility will be put together first, but little testing needs to be done directly on it, since it will be pulled right out of Sun's XDR code, with minimal adaptation. The next piece to be built will be the HTTP+TCP MTL (since this can mostly be ripped out of our prototype), and then a test client and server will be built to test it. The client will send data patterns of various sizes, and the server will just send them back. Then the eStream Procedure Call PCL will be built and tested with another test client and server at this level. Finally the SSL+TCP MTL will be constructed and tested with the null authentication/encryption/hashing configuration, which will still test basic handshaking. Once other security issues are handled, we'll enable full SSL functionality.

Beyond the basic testing, certain features need specific testing: client reply timeouts [put sleeps in server remote procedures and also just have servers not respond at all], server connection idle timeouts [put sleeps in clients], transparent connection re-establishment [just have a client reuse the connection from above], connection states [need to start making lots of requests while a connection is being asynchronously started], message states [different amounts of stuff on the queues in & out], reuse of connection objects.

Then error conditions need to be tested, since if EMS hangs, the whole system is effectively down. Evil test clients will drop connections, even get killed, send garbage data with bad headers (various possible fields), send huge messages. Of course, implementers of EPC functions need to sanity/validity check any incoming parameters of those functions. Evil test servers will respond with bad reponses as well. Coverage tools will likely come in handy, since much of EMS's code will be present to check for all kinds of errors.

Stress testing plans

Stress testing is very important for EMS. To satisfy this need, I'll need a server that implements remote procedures for the various test cases above, across more than one EPC API, and then run a bunch of the above singular clients all at once, with multiple instances of each. That will stress EMS on the server side, and should be run for many minutes (along with evil test clients) to try to expose any memory leaks and other corruption. Additionally, I'll build a client that communicates with multiple servers, basically taking the above test clients and running them in different threads in the same process, thereby stressing the single shared EMS code on the client side.

Cross-component testing plans

EMS's interaction with other components that needs to be tested besides the external interface is its interaction with proxy servers and firewalls. We'll have to implement some kind of test bed that resembles a real world setup and run EMS through it, perhaps with different potential configurations. We needs somebody familiar with proxy servers and firewalls to look at this.

Upgrading/Supportability/Deployment design

EMS will report errors to the caller as they occur (and log them if they're serious). Some errors may be fatal to the component in which case it'll have to do the right thing (do we have a common "out of memory" handler or something like that we care about?). Of course for debugging, there will be debug logging, but I'm not sure we want to put runtime checks for that in or not.

There are going to be rules and assumptions for the EMS layers that I'll have to describe so that future additional layers will conform and thus integrate properly. I'll add these to the document here once those rules are solid, which won't happen until most of the implementation is complete.

Open Issues

1. Need to figure out how to integrate turning off TCP send coalescing (or see if SSL already uses it).
2. How do shutdown of EMS system? (wait for EPC worker threads, etc?)
3. Still need to think about how newly formed connections work on the server. Likely only to fully understand once implementation is underway.
4. RealPlayer can get the proxy address from the local installed browser (so it claims), probably from the registry. Maybe we should do this too? Does this apply to SSL proxies as well?

Server Installation Requirements
Author: Bhaven Avalani
Omnishift Confidential

Pre-requisites:

1. Hardware:
   - 3 dual-processor machines with the following configurations:
     i. 400 Mhz (or higher) dual–processor CPU.
     ii. 128 MB (or higher) RAM.
     iii. 20 GB (or higher) disk space.
     iv. 10 GB (or higher) disk space on C: drive.

   (We shall call these machines A,B and C for further discussions).
2. Software:
   - Each of the machines should be loaded with Widows 2000.
   - Machine A to be configured with SQL Server 7.0.
   - Machine B to be configured with Apache(1.3.12) webserver and Tomcat(3.1) servlet engine.
   - Machines A,B and C to be configured with SQL Server(7.0) clients.
3. Software shipped by Omnishift:
   - Web Server Software:
     i. DDL script to create the EStream data model.
     ii. webserver.jar (Servlets to access the eStream database).
     iii. Sprinta2000.jar (JDBC driver from Inet software).
     iv. JSP and HTML pages for the Web server applications.
   - AppServer.exe
   - SlimServer.exe
   - Monitor.exe
   - DLLs:
     i. otdbdll.dll (Dll for ODBC database connectivity).

Installation:

1. The software provided by Omnishift will be in a zip format (for Alpha). **Unzip** the Omnishift software. This will create the following subdirectories:
   a. C:/eStream1.0/bin (All the exes and dlls will go here).
   b. C:/eStream1.0/logs
   c. C:/eStream1.0/conf (to store the flat file configurations).
   d. C:/eStream1.0/ess (to store the estream sets).
   e. C:/eStream1.0/esc (to store the estream cache).
   f. C:/eStream1.0/web (to store all web server related components).
2. Create the eStream database using the data model script supplied on Machine A.
3. Configure the eStream ODBC data source on each of the machines.
4. Configure the Apache/Tomcat server with eStream software.
5. Configure the physical machines using the Admin UI.
6. Configure the monitor services on machines A, B and C.
7. Configure the monitor setting in the database using the Admin UI.
8. Configure Slim Servers on machine A and C.

9. Configure App servers on machine B and C.
10. Deploy the eStream sets on C:/eStream1.0/ess directory.
11. Start the Slim and the App servers.

Development setup:

1. Install *Visual Café* and *Dreamweaver* from \\wkst18\download\WebGainStudio40(Trial).exe or http://www.webgain.com/.
2. Install *Tomcat 3.1* from http://jakarta.apache.org/builds/tomcat/release/v3.1/bin/.
3. Install *SQLServer 7.0*.
4. Install JDBC driver from inet software. \\wkst18\download\sprinta2000_trial.zip.
5. Install JDK 1.3 from http://java.sun.com/j2se/1.3/.
6. Set environment variable TOMCAT_HOME and JAVA_HOME to your Tomcat and JDK root directory (e.g. c:\tomcat and c:\jdk1.3).
7. Add JDBC driver path (e.g. c:\JDBCDriver\Sprinta2000.jar) and eStream application path (e.g. c:\tomcat\webapps\classes\webserver.jar) to your CLASSPATH.
8. Add eStream application into tomcat's server.xml file (e.g. add "<Context path="/estream" docBase="webapps/estream" debug="1" reloadable="true" ></Context>" with other sample applications in c:\tomcat\conf\server.xml file).
9. In Project\Option of *Visual Café*, point the deployment directory and path to your tomcat's eStream directory and file (e.g. c:\tomcat\webapps\classes and webserver.jar), and put JDBC driver path into project directories (e.g. c:\JDBCDriver\Sprinta2000.jar).

Server Installation Requirements
Author: Bhaven Avalani
Omnishift Confidential

Pre-requisites:

1. Hardware:
   - 3 dual-processor machines with the following configurations:
     i. 400 Mhz (or higher) dual–processor CPU.
     ii. 128 MB (or higher) RAM.
     iii. 20 GB (or higher) disk space.
     iv. 10 GB (or higher) disk space on C: drive.

   (We shall call these machines A,B and C for further discussions).
2. Software:
   - Each of the machines should be loaded with Widows 2000.
   - Machine A to be configured with SQL Server 7.0.
   - Machine B to be configured with Apache(1.3.12) webserver and Tomcat(3.1) servlet engine.
   - Machines A,B and C to be configured with SQL Server(7.0) clients.
3. Software shipped by Omnishift:
   - Web Server Software:
     i. DDL script to create the EStream data model.
     ii. webserver.jar (Servlets to access the eStream database).
     iii. Sprinta2000.jar (JDBC driver from Inet software).
     iv. JSP and HTML pages for the Web server applications.
   - AppServer.exe
   - SlimServer.exe
   - Monitor.exe
   - DLLs:
     i. otdbdll.dll (Dll for ODBC database connectivity).

Installation:

1. The software provided by Omnishift will be in a zip format (for Alpha). Unzip the Omnishift software. This will create the following subdirectories:
   a. C:/eStream1.0/bin (All the exes and dlls will go here).
   b. C:/eStream1.0/logs
   c. C:/eStream1.0/conf (to store the flat file configurations).
   d. C:/eStream1.0/ess (to store the estream sets).
   e. C:/eStream1.0/esc (to store the estream cache).
   f. C:/eStream1.0/web (to store all web server related components).
2. Create the eStream database using the data model script supplied on Machine A.
3. Configure the eStream ODBC data source on each of the machines.
4. Configure the Apache/Tomcat server with eStream software.
5. Configure the physical machines using the Admin UI.
6. Configure the monitor services on machines A, B and C.
7. Configure the monitor setting in the database using the Admin UI.
8. Configure Slim Servers on machine A and C.

9. Configure App servers on machine B and C.
10. Deploy the eStream sets on C:/eStream1.0/ess directory.
11. Start the Slim and the App servers.

Development setup:

1. Install *Visual Café* and *Dreamweaver* from \\wkst18\download\WebGainStudio40(Trial).exe or http://www.webgain.com/.
2. Install *Tomcat 3.1* from http://jakarta.apache.org/builds/tomcat/release/v3.1/bin/.
3. Install *SQLServer 7.0*.
4. Install JDBC driver from inet software. \\wkst18\download\sprinta2000_trial.zip.
5. Install JDK 1.3 from http://java.sun.com/j2se/1.3/.
6. Set environment variable TOMCAT_HOME and JAVA_HOME to your Tomcat and JDK root directory (e.g. c:\tomcat and c:\jdk1.3).
7. Add JDBC driver path (e.g. c:\JDBCDriver\Sprinta2000.jar) and eStream application path (e.g. c:\tomcat\webapps\classes\webserver.jar) to your CLASSPATH.
8. Add eStream application into tomcat's server.xml file (e.g. add "<Context path="/estream" docBase="webapps/estream" debug="1" reloadable="true" ></Context>" with other sample applications in c:\tomcat\conf\server.xml file).
9. In Project\Option of *Visual Café*, point the deployment directory and path to your tomcat's eStream directory and file (e.g. c:\tomcat\webapps\classes and webserver.jar), and put JDBC driver path into project directories (e.g. c:\JDBCDriver\Sprinta2000.jar).

eStream 1.0 Low Level Design

Software License And Management (SLiM) Server

Amit Patel
Last Modified: [redacted]
Version 2.0

Functionality

The Software License Management (SLiM) server is required to enforce licensing terms and track overall application usage. Its primary function is to grant, renew and expire access tokens, record application usage and aid in server load balancing. Its design can be broken down into three somewhat orthogonal axis:

1. Detailed specification of eStream client interfaces – the need.
2. Design and usage of Server Common Services (CSC) & server database –the tools of the trade. These include logging, system monitoring, thread package, encryption, TCP/IP communications, etc.
3. Core SLiM server logic that fulfils client interfaces (1) using CSC (2).

This document address items 1 and 3 in detail; item 2 should be covered in various other documents emerging from the server team.

Data type definitions

Common Data Types

- Standard atomic data types everyone (clients, builders, servers) must agree on: **Int8, Int16, Int32, Int64, uInt8, uInt16, uInt32, uInt64, uInt128** (specially for GUIDs).
  - Notice: No floating point types; I don't see a compelling reason to pass floating-point number across wire.
- **String** is represented as a size (uInt32) and it contents. Its contents must include a NULL termination character, it must be included as part of the size field.

Length	characters.....

- All eStream sets will use little endian representation.
- All complex data structures between major components (def: at least client/servers) should be version identifiable. Proposal: put a version number (uint32) as 1st word in the structure.
- Most complex structures are variable length because they contain strings. I think it would be good to put the total size (in bytes) of that structure as second word so that reader can know how much to read. If marshalling/unmarshalling utilities provide a way to represent that, it won't be needed in each struct.
- We'll need a single place where all globally (definition: across client and servers, and between servers) visible macros (#define & enum) are defined. We'll also need reserved

name spaces, and reserved number ranges for different components. Until all that is decided, I am defining macros without assigning any values.

- I am assuming that our message pack/unpack utilities will deal with alignment issues.

Server Sets

EstreamServerID contains server parameters clients need to know in order to initiate a connection. EStreamServerSet is simply a list of individual server Ids. The main use of this data structure is for SLiM server to return a list of app servers that can serve a given application. Server ids and server sets are specific to each application; client is responsible for keeping a map of app id → server set.

```
#define     SERVER_TYPE_APPLICATION
#define     SERVER_TYPE_SLIM
#define     SERVER_TYPE_ASP_WEB

typedef struct {
     uInt32     Version;
     uInt32     SizeInBytes;
     uInt32     MachineIP;
     uInt16     MachinePort;
     String     MachineName;
} eStreamServerID;

example of an eStreamServerID:
    {1,20,0x12348,80,{12,"s10.asp.com"}}

typedef struct {
     uInt32              Version,
     uInt32              SizeInBytes,
     uInt32              ServerType;
     uInt32              ServerIDCount,
     eStreamServerID     ServerID[];
} eStreamServerSet;

example of an eStreamServerSet:
{1,??, SERVER_TYPE_APPLICATION, 2,
     {1,20,0x12348,80,{12,"a10.asp.com"}},
     {1,20,0x12349,80,{12,"a11.asp.com"}}}
```

Access Tokens

This is a main data structure that is getting passed back and forth between a client and SLiM/App servers. Granting an AccessToken is an acknowledgement of client's legal right to run the application – the license. Denying an AccessToken is an acknowledging that the client does not have rights to run the application; probable causes include a user running multiple sessions, user not paying bills etc.

From client's perspective, it is totally opaque; but SLiM server uses it to pass information to the app servers so that the app server does not have to rely on the database lookups. Each access token will have a unique ID.

Terminology

Billing Granularity – Granularity at which an ASP is interested in billing its customers. Most ASPs today bill on monthly basis and eStream will assume that to be the 'norm'. However, to support things like short trial memberships, we'll design eStream to handle billing as often the AccessToken Renewal Frequency (defined below). If an end user simply purchases the eStreamed application, the billing granularity is infinite – the upper bound. EStream should not assume that billing granularity for all apps served by an ASP is the same.

AccessToken Renewal Frequency – Frequency at which the client must renew its access tokens in order to continue eStream application use. This must be tunable parameter whose upper bound is the billing granularity; it is also the smallest billing granularity we'll support. Not all access tokens are required to have the same renewal frequency.
Recommendation: **10 minutes**.
Tradeoffs:

1. This is the smallest granularity at which a client can be evicted (defined below).
2. Finer granularity may increase the number of hits to the SLiM server and adversely effect its scalability.

Eviction Notice – In general there will be times when an ASP wants to stop a user from using an eStream application, which also means stopping a user from consuming ASP server resources. Possible reasons may be:

- Lack of payment.
- Termination of a trial membership.
- To force the client into upgrading an app.
- Just because the restroom is freezing cold.

EStream infrastructure has an inherent limitation that servers can't push anything on the client. That means SLiM servers must deny an access token or its renewal, to effectively deliver an eviction notice to the client. Also, App servers may need to be informed of such evicted access tokens so that they can deny paging requests.

Decision: *After looking at some scalability numbers, we concluded that a renewal frequency of 10 minutes should not affect the overall performance and scalability of eStream system. Consequently, we don't have to communicate the list of evicted tokens to the app server since they would be invalid soon (avg 5 minutes) anyways. This simplifies server designs by reducing cross communication between slim servers and app servers.*

```
typedef struct {
  uInt32      Version;
  uInt32      SizeInBytes;
  uInt128     ATID;          // AccessToken ID - GUID
  String      UserId;        // GUID.
```

```
    uInt128     AppID;          // GUID.
    uInt64      IssueTime;      // POSIX time_t format.
    uInt64      ExpirationTime;
} eStreamAccessToken;
```

Other Common Data Structures

In order to allow easy/automatic updates of eStream application, we need to define a protocol by which a client can be informed of app updates. This structure will also be used when installing subscribed applications on a client.

AppName, VersionName – describe the application.
Message – a short description of an application.
Flages, such as ForcedUpgrade – client must upgrade the application.
RootFileNumber - is sort of the version # of an application root directory.
RootFileMetadata - metadata of the root directory.

```
typedef struct {
    uInt32      Version;
    uInt32      SizeInBytes;
    uInt128     AppID;
    String      AppName;        // may be "Word2000"
    String      VersionName;    // may be "SP1"
    String      Message;
    uInt8       ForcedUpgrade;
    Int32       RootFileNumber;
    ????        RootFileMetadata;

} eStreamAppInfo;
```

Interface definitions

In a single process context, cross-module interfaces are easy and intuitive when defined as C/C++ procedure calls. However, for client/server (and perhaps server/server) interfaces, we need to define our own RPC-like protocol. Sameer covering this (EMS – estream messaging services) in a different design, but I want to state couple of assumptions I am making:

- Each EMS call is assigned a unique number (Int32). Codes must be uniform across all servers (i.e. no duplication of names and numbers). We should reserve some namespaces and numbers for each eStream server. Following is the current list of EMS codes between SLiM/Clients.

```
#define     EMCC_NULL
#define     EMCC_ACQUIRE_ACCESS_TOKEN
#define     EMCC_RENEW_ACCESS_TOKEN
#define     EMCC_RELEASE_ACCESS_TOKEN
#define     EMCC_REFRESH_SERVER_SET
#define     EMCC_GET_LATEST_APP_INFO
#define     EMCC_GET_SUBSCRIPTION_LIST
```

- In addition to any data (pages etc.), an EMS calls needs to return a number of codes to communicate success/errors. Following structure provides a container for returning multiple return codes. By convention, we'll put either EMCR_FAILURE or EMCR_SUCCESS in Code[0].

```
typedef struct {
        uInt32      SizeInBytes;
        uInt32      ReturnCodeCount;
        uInt32      ReturnCodes[];
} EMCReturnCodes;

#define     EMCR_SUCCESS
#define     EMCR_FAILURE
#define     EMCR_USER_AUTH_FAILED
#define     EMCR_ACCESS_TOKEN_INVALID
#define     EMCR_SUBSCRIPTION_INVALID
#define     EMCR_LICENSE_NOT_AVAILABLE
#define     EMCR_LICENSE_ALREADY_HELD

#define     EMCR_EVICTION_NOTICE
#define     EMCR_EVICTION_MUST_UPGRADE
#define     EMCR_EVICTION_END_MEMBERSHIP
#define     EMCR_EVICTION_NO_PAYMENT
```

Acquire Access Token

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_ACQUIRE_ACCESS_TOKEN	
IN	uInt128	SubscriptionID
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamAccessToken	AccessToken
OUT	uInt32	RenewalFreq
OUT	eStreamServerSet	AppServerSet
OUT	eStreamAppInfo	LatestAppInfo

Client will use this interface prior to starting an eStream application to grab the license. It accepts a subscription id, which clients received when an app was subscribed, and password; it replies with at least a list of return codes and possibly, the access token, its renewal frequency and a set of servers that can serve this app.

AccessToken – Client treats them as opaque data structures and renews them within its renewal frequency.

RenewalFreq – This uInt64 is the number of seconds the access token is valid for once it receives it. You probably don't want an absolute count (i.e. # of seconds since epoch) since clients can interpret it differently due to clock skew.
ServerSet – When a client gets an access token, it will be given a list of app servers that can serve the particular app. The ServerType member of eStreamServerSet structure will be SERVER_TYPE_APP. The list is specific to each app and should be managed as such.
LatestAppInfo – SliM servers will pass information about the latest app version (root) using this structure. Refer to client eFS design for more detail. This structure will always be passed; client will ignore it if it already has the latest version.

> Note: There is a big difference between major and minor upgrades: a major upgrade would be going from word 98 to work2000 (where app ids must change) where as a minor upgrade (app ids will not change) means applying a patch or a service pack. LatestAppInfo tries to transparently propagate latter (minor upgrades) to end users without requiring end users to unsubscribe/subscribe apps. Major upgrades will require end users to go back to the ASP web server and change subscriptions. ASP can force the end user into changing subscriptions (word 98 to word 2000) using EMCR_EVICTION_MUST_UPGRADE error code.

ReturnCodes
Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED – Can't authenticate user with specified passwd.
- EMCR_LICENSE_NOT_AVAILABLE – License is not available.
- EMCR_LICENSE_ALREADY_HELD – If the user is already holding the license, SLiM server returns this error code along with the access token that is held & its renewal frequency. Most common cause of this error is when an end user tries to run an eStream app on two different machines simultaneously. NOTE: returned token doesn't give the right to run the application and should be treated as a denial of access token. Reason for returning the token/renewal interval is to allow the client software can effectively release the token, wait some time (>= renewal frequency) and re-try.
  - The reason client has to wait is because SLiM servers will not communicate the list of 'bad' access tokens to the app server.
- EMCR_EVICTION_NOTICE –ASP wants to stop the user from using ASP resources. Server may also add code that describe the reason like 'no payment' etc. Note that no access token will be given! This may change in the future to allow some grace period.
  - EMCR_EVICTION_MUST_UPGRADE – This type of eviction means the ASP wants the end user to stop using this particular application in favor of another (major) version of it. For example, Word 98 to word 2000.

Renew Access Token

Caller:	eStream Client	
Callee:	SLiM Server	
RPC Code:	RPCC_RENEW_ACCESS_TOKEN	
IN	String	UserName
IN	String	Password

IN/OUT	eStreamAccessToken	AccessToken
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamServerSet	AppServerSet
OUT	uInt32	RenewalFreq

Clients will use this interface to renew the access token before it expires. Client will specify the old access token and if there are no errors, get back EMCR_SUCCESS, a new access token, new app server set (ServerType field of eStreamServerSet structure will be SERVER_TYPE_APP) and new renewal frequency. Upon getting the new app server set, client **must** remove the old app server set for this application. If for some reason, the access token is expired, SliM server will treat this request as 'Acquire Access Token' and may return error codes possibly from that interface (this is one of the reason for asking for usernames/password).

NOTE: Unlike Acquire Access Token, it is not returning LatestAppInfo or EMCR_EVICTION_MUST_UPGRADE error codes because once the app is running, we can't upgrade apps while running.

ReturnCodes
Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID – Access token is invalid.
- EMCR_EVICTION_NOTICE –ASP wants to stop the user from using ASP resources. Server may also add code that describe the reason like 'no payment' etc.
  - NOTE: will NOT return EMCR_EVICTION_MUST_UPGRADE.
- Error codes from Acquire Access Token.

Release Access Token

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_RELEASE_ACCESS_TOKEN	
IN	eStreamAccessToken	AccessToken
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes

Client will use this interface to release the license held by the specified user. It should be called synchronously when the application exits or crashes. The reason for requiring usernames and password is to authenticate the identity of the caller against access token owner. The reason for proactively releasing tokens as opposed to just letting them expire is because releasing it allows the user to re-acquire it (on the same or different machines) without waiting for it to expire. This allows the user to do acquire -> release -> acquire without any wait.

ReturnCodes
Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID
- EMCR_USER_AUTH_FAILED

Refresh App Server Set

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_REFRESH_APP_SERVER_SET	
IN	eStreamAccessToken	AccessToken
IN	uInt8	BadQoS
IN	uInt8	NoService
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamServerSet	ServerSet

App Server sets are given to a client when an access token is acquired and are automatically refreshed when an access token is renewed. However, the client can always refresh its app server sets using this interface. Potential reasons for clients to do this:

- All servers in the current server set are not responsive – NoService = TRUE
- Servers are up, but client experiences bad QoS (network delays/timouts). BadQoS = TRUE

ReturnCodes

Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID

Get Subscription List

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	EMCC_GET_SUBSCRIPTION_LIST	
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes
OUT	uInt32	NumberOfSubscriptions
OUT	uInt128[]	SubscriptionID[]

A client can ask for the current list of subscribed applications using this interface. SLiM server returns the number of apps subscribed and an array of subscription ids.

ReturnCodes

Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED

Get Latest Application Info

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_LATEST_APP_INFO	
IN	String	UserName
IN	String	Password
IN	uInt128	SubscriptionID
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamAppInfo	UpgradeInfo

Any upgrades pending? This functionality is piggy backed on 'acquire access token' interface, but there is some value in providing it as an explicit interface. SliM server will give you the latest application information block associated with the specified subscription id; the client can decide if it already has the latest root (version) or not.

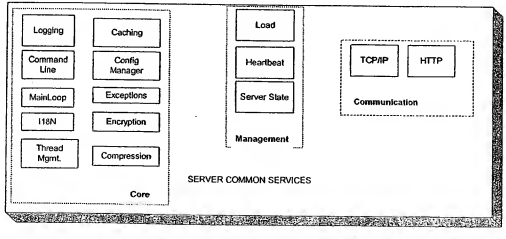
ReturnCodes
Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED
- EMCR_SUBSCRIPTION_INVALID

Component design

Server Common Services

Following diagram shows the common portion of all eStream servers. Most of these boxes won't be described in this document because they are covered in specific documents.



Various Decisions

- SLiM server will uses an ODBC interface to communicate with the central database.

- From eStream client's perspective, all SLiM servers are equal in functionality. This is unlike an application server, which can be segmented to serve specific applications.
- Each SLiM server will have a unique IP/Port combination. Multiple SLiM servers running on the same machine can be distinguished by giving them different port numbers.
- SLiM servers (and app servers) will not assume any default ports; it will rely on an ASP admin to configure the port assignment. With help from OTI, ASP admin will determine how many eStream servers need to run on a machine and assign a unique number to each eStream server.

Hardware Failover

There will be a pool of SLiM servers at an ASP site; from eStream client's prospective, each of them is identical. When a client subscribes to an eStream application, it gets a set of SLiM servers to communicate with. The clients will keep this list in memory and refer to it when calling SLiM server interfaces. If it experiences difficulty communicating with a particular SliM server, it will try other servers that are part of the server set. If for some reason the server set is lost, or all servers in the set are not responding, a client can always go back to the ASP web server and refresh its server set. This gives you a transparent (from end user's prospective) hardware fail over path.

The same approach will also work for app server fail-over scenarios; specific differences are that:

1. SLiM Servers, not ASP web servers, will provide the app server set.
2. App server list will be refreshed automatically, when access tokens are renewed. This allows ASP admins to take out servers from the pool by waiting certain amount of time (>= access token renewal frequency) and not cause unnecessary client timeouts.
3. App server sets are specific to each app; SLiM servers are not.

Load Balancing

In eStream 1.0, we will not require a third party load balancers at an ASP site; we'll do minimal things at both ends (clients/server) that should be good enough for small to medium size ASPs. We may have to test with selective 3rd party load balancers to see if we can work with them or not; but this is an open issue (listed in the open issues at the end of doc.).

In eStream 1.0, we'll capitalize on the hardware fail-over mechanism to also aid load balancing. Following two actions will perform load balancing:

- When a client gets server sets (app or SLiM servers), it will distribute its hits randomly among the server in the set. In addition, clients will also get new app server sets every time they renew access tokens.
- On server side, the monitor will keep track of each server's response times to process client's requests. The data gets sorted from most responsive to least responsive and stored into the database. Top 'X' servers from this list will be given to the client when it makes an explicit request to refresh its app server set, acquires or renews an access token.

Testing design

Interface Testing

SLiM server is tightly coupled with three components: client, ODBC/Database and monitor; it is fairly difficult to isolate it from *all* of those components for unit testing. A better approach is to exhaustively test the client/SLiM server interfaces, which will in fact also test large numbers of interfaces to the other two components. The idea is to crank up a client that will make every possible SLiM server request and make sure that SLiM server responds accordingly.

I think it is good idea to create a simple testing framework (that may evolve with time) that will simulate a real client to SLiM and app servers. We can do this by writing a program that includes common (client/server) data structures definitions, links in our eStream Message Services (EMS) component and invokes various interfaces like 'Acquire Access Token'. From SLiM server's prospective, this test program is a working client.

For each client/server interface (i.e. Acquire access token) write a test case (dummy client) that will:

- Assume that we have created a dummy database that has certain users, passwords and subscriptions.
- Invoke SLiM server with all possible input permutations. This isn't too bad since most interfaces have 2 to 4 arguments.
- In the process, ensure that SLiM server returns all possible return values it can.

For instance, lets assume that Acquire Access Token has following prototype:
AET(uInt128 subID, String UserName, String Password);

```
TEST BEGIN:
    Assert (AET(NULL, NULL, NULL) returns
            EMCR_FAILURE & EMCR_USER_AUTH_FAILED);

    Assert (AET(good_sub_id, good_user_name, good_password) returns
            EMCR_SUCCESS, an access token, its renewal freq. Etc.);

    Assert (AET(good_sub_id, good_user_name, good_password) returns
            EMCR_FAILURE & EMCR_LICENSE_ALREADY_HELD);
```

Stress testing plans

Stress testing in general will be common across all eStream servers. I think it will be a good idea to invest in a 3rd party tool that can simulate real-time load on eStream servers and see its responses. Rational has various tools such as Visual Test, Robot and Site Load that are worth evaluating.

Coverage testing plans

Lot of these items will apply to all eStream components and probably should be covered in a separate eStream test plan document. I am not sure if we should do these things before each component is done or wait until they are integrated. I just want to state what may be obvious so that it is documented:

- SLiM server will achieve 85% PFA coverage as measured by Rational PureCov. Tests used to measure PFA coverage will be reproducible, either by hand or via an automated test suite.
- SLiM server will resolve all memory corruption and memory leak issues as reported by Rational Purify.
- We should have test cases that will exercise all command line options for SLiM server.
- SLiM server will be code reviewed by at least two peers.

Upgrading/Supportability/Deployment design

This document must have a discussion of how the component addresses any specific issues related to upgrading, supporting and deployment of e-stream applications. Some examples include: error conditions detected and reported by this component, any special hooks this component will provide for monitoring, hints for troubleshooting problems, any special hooks for debugging this component.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

1. How do you produce GUIDs on unix servers? Should app ids, user ids, access token ids be guids or we should create them by knowing what numbers are already used?
2. Resolve big-endian – little-endian issues. Owner: Sameer
3. Meaning of 'eviction' notice is not conveyed to the end user yet. Owner: Client person – Ann.
4. Encryption impact on SLiM servers. Owners: Amit & Igor.
5. Global name space & number ranges for different components. Owner: Bhaven
6. ASCII v/s Unicode strings? Owner: Sameer.
7. Test with 3rd party load balancers to see if we work or not. Requirements for deployment team: tell us which load balancer to certify against and set them up in our future testing lab. Owner: deployment team.

eStream Web Server Load Monitoring Applet Low Level Design

Jae Jung

Modified [redacted]

Functionality

One of the requirements for the eStream web server is a facility for monitoring server load (eStream requirements 3.0, 3.2, 3.4, 3.5). Per this document, this facility will be provided by a graphical load-monitoring applet that will be available for deployment at customer sites as part of the eStream web server installation.

The load-monitoring applet will present information in the following formats through a graphical interface:

- Real-time server load information plotted on strip chart
- Historical server load information plotted on line chart
- Multi-server real-time or historical load information on a line chart

Requirements

The following list details the provisional requirements for the load-monitoring applet. The remainder of this design document is based on these requirements.

1. The applet will be able to display server loads in "real-time" as load data is retrieved from the server.

2. The applet will be able to display (in a separate mode from real-time monitoring) historical load information to the extent that this information is available in the database.

3. The applet will be configurable (via applet parameters) for the following settings:
   -- Data retrieval rate, i.e., the frequency with which the applet request new load data from the web application server
   -- Chart window size, i.e., the number of datapoints shown in the chart window at any one time.

4. The applet will be capable of concurrently displaying the load of multiple servers in a clear and concise fashion.

5. The applet will support the displaying of cumulative and average load statistics for multiple servers.

6. In real-time mode, the applet will operate as a strip chart, with the fastest chart speed determined by a global configuration setting in database, typically corresponding to the frequency with which the eStream Monitor inserts load records into the database.

7. The applet will retrieve load information from the database via an http connection to the eStream web app server.

8. The applet will run in browsers with Java 1.0.x and 1.1.x support.

9. Internationalization support. Both the Applet and the backend pieces should be internationalizable.

Description

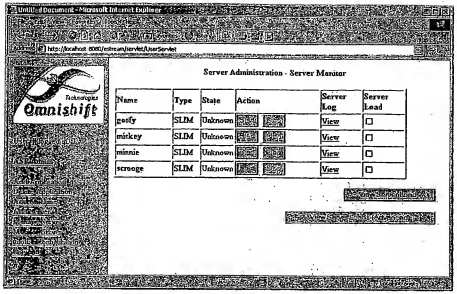
The load monitoring applet is comprised of two components. The first component will manage the retrieval of real-time or historical server load information from the eStream database. The second will consume this data and present this information to the user in a clear and concise graphical format. In addition to the applet, server-side objects will need to be written or extended to service data requests from the applet.

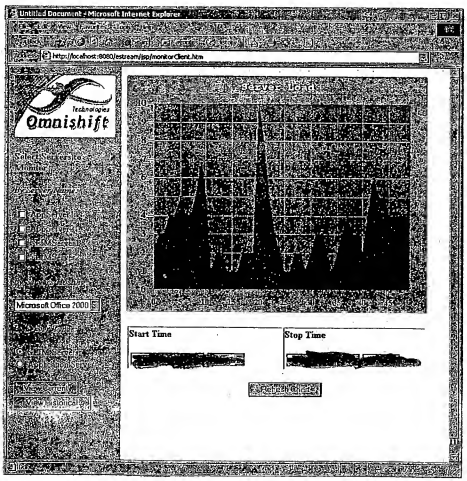
For the alpha-release of the eStream web server, the graphical presentation component will not be written internally; rather a commercially available applet or package will be used. Other aspects of the applet and the server-side components will be implemented to facilitate transitioning to an internally developed graphical component if such a decision is made for later releases.

User Interface Design

The following two screen shots are representative of the way that the load monitoring applet might be used in a particular monitoring/administration Brower interface. The first shot shows a general server administration and monitoring UI and the second shows a detailed load monitoring UI within which the load monitoring applet will be embedded.

The UI options shown in the second shot illustrate some of the reporting options for which the load monitor will be initially configurable. The applet(s) will be readily extensible to generate additional reports and more complex data combinations if desirable.


Untitled Document - Microsoft Internet Explorer
http://localhost:8080/estream/servlet/UserServlet
Omnishift
Server Administration - Server Monitor
Name
Type
State
Action
Server Log
Server Lead
geofy
SLIM
Unknown
View
mickey
SLIM
Unknown
View
minnie
SLIM
Unknown
View
scrooge
SLIM
Unknown
View


Untitled Document - Microsoft Internet Explorer
http://localhost:8080/estream/jsp/monitorClient.htm
Omnishift
Server Monitor
Start Time
Stop Time

Interface Definitions

The load monitoring applet has two interfaces with other webserver components: <APPLET> tag parameters for its configuration within the web browser page and the HTTP request/response format with the web application server to request and receive load data.

1. **Applet <-> HTML Page Interface (<APPLET> Tag Parameters):**

Parameter	Req'd?	Significance	Default
hostName	Yes	Host name of webserver	None
hostPort	No	Host port of webserver	"80" (int)
chartSpeed	No	Time (in seconds) between chart scroll; also the resolution of the x-axis	"5" (int)
updatePeriod	No	Interval (in seconds) between HTTP requests for server load data. Note if chartSpeed < updatePeriod, the applet buffers load data for presentation. Maximum information latency will at most be equal to this value plus round trip time for the data request.	"10" (int)
chartWidth	No	Width (in ticks) of the applet chart; in conjunction with chartSpeed, implicitly determines the amount of data displayed	"50" (int)
mode	No	"current" for real-time monitoring and "history" for historical	"current"
startDate	No	if mode="history", the starting date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored.	none; value must be in Java Date format
stopDate	No	if mode="history", the stop date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored. Note that stopDate override the chartSpeed setting insofar as x-axis resolution is concerned.	none; value must be in Java Date format
numServers	Yes	number of servers to be plotted	1 (int, max 50)
serverName1 serverName2 ... serverName50	Yes	the id of the server(s) being plotted (up to 50 servers can be plotted at once). This must correspond to a existing serverID in the Server database table	none
serverData1 serverData2 ... serverData50	No	an initial set of data with which to display the initial chart. Note that these parameter(s) are the default means by which historical load data is displayed. Each set should be a comma delimited list of numbers (float)	none

- Note that applet tag parameters are all strings; where the applet does an internal type conversion, the target type format of the particular parameter has been noted.

- Note that the parameters determining the appearance of the chart (e.g., line colors, grid painting options, custom labels, etc.) are not included in this list. These parameters will either be determined by the parameters available for configuration in the commercial charting applet or TBD at some later date if the charting component is developed internally.

2. Applet <-> Web App Server Interface (HTTP)

Input:

The load monitoring applet will request load data from the webserver by executing the web server's MonitorServlet with the following parameters:

href="/MonitorServlet?action=getLoadData
&serverId=...
&startDate=...
&stopDate=...
&numPoints=...

where:

- serverId is a comma-delimited list of one or more server(s) for which load information is being requested. Note that this serverId corresponds to the serverID attribute in the Server database table.
- startDate is the (exclusive) starting date (in Java Date format) for which the load information is being requested
- stopDate is the (inclusive) end date (in Java Date format) for which the load information is being requested
- numPoints is the number of data points requested between the start and stop dates.

The applet will process the return data points as equally time-spaced points between the requested start and stop dates.

In addition, if startDate=stopDate, only one load data point (i.e., the most recent) will be returned by the servlet.

Output:

The load monitoring applet will expect load information from the web server via HTTP response in the following XML-like format:

<loadData>

```
<serverid=a >
        <LOADLIST>
                <LOAD value=x1/>
                <LOAD value=x2/>
                <LOAD value=x3/>
            ...
                <LOAD value=xN/>
        </LOADLIST>
</server>
<server id=b>
<LOADLIST>
                <LOAD value=y1/>
                <LOAD value=y2/>
                <LOAD value=y3/>
            ...
                <LOAD value=yN/>
        </LOADLIST>
</server>
    ...
</loadData>
```

In the above example, x1 to xN represents load values for the server a (by serverID), and y1 to yN represents load values for server b.

Testing Design

The load monitoring applet and related server-side Java code will need to be tested according to the plan outlined for other web server components in the Web Server/Database Low Level Design (WebServerDB-LLD.doc). Additionally, it should be noted that certain applet-related parameters (i.e., updatePeriod) that affect the frequency with which the applet requests load data from the web server are good candidates for tuning in order that a good balance between UI/measurement response and web server response performance be struck.

Upgrading/Supportability/Deployment Design

Upgrading/Supportability/Deployment Design will follow the model described for the Web Server in the Web Server/Database Low Level; Design (WebServerDB-LLD.doc).

Open Issues

1. How much will it cost to deploy the chosen commercial java applet/package(s) as an OEM installation? We need to find this out before we decide on a commercial

package. Also, another criteria for the choice would be: will we get support. Do we get the source code too?

2. Longer term, where's the cost/benefit breakpoint for the above where it makes more sense to write our own charting applet/package?

eStream Web Server/Database Low Level Design

Bhaven Avalani
[redacted]
Modified [redacted]

Functionality

The eStream solution provides a set of account, user, and subscription management utilities. These utilities are provided as extensions to the ASP's(Application Service Provider) web server.

There are three categories of users for these utilities: End User, Group Administrator and ASP Administrator. The roles and the capabilities of each of these users are detailed below.

End user for a system is the user who will actually access eStream application using the eStream clients. An end user should be able to:

- Create Account and User attributes. (Username, Password, etc.)
- Change Account and User attributes.
- View all available applications in the eStream system.
- Subscribe/Manage eStream applications.
- View Account Status.
    1. List of applications subscribed.
    2. Status of current subscription.
    3. View/Change Billing information.
    4. View/Change Account information.

A *Group Administrator* is an administrator for a group of users. An individual user is by definition a group administrator for a single user group. Capabilities of a group administrator are:

- (All of single user capabilities).
- Add delete users from a group.
- Manage the active sessions for a group. A group manager should be able to release licenses from active sessions, thereby kicking out active users.
- View the billing information. This will probably need hooks to an external billing system.

An *ASP administrator* manages the overall application system. Capabilities of an ASP administrator are:

- Manage accounts/users/subscription for all users/groups in the system.
- Manage the application data for a subscription system.

1. Add new applications to the system.
2. Modify application information for the system.
3. Provide the pricing mechanism for the applications(?).

- Manage the servers in the system.
  1. Configure a server.
  2. Stop/Start a server. This is accomplished by a message to the Monitor server.
  3. Get load information for a server.
  4. Get logging information for a server.

There are essentially two different types of accounts, which the system will support: Single user account and corporate accounts.

The following licensing mechanisms will be supported by the system.

- Fixed Duration License. (Typically monthly license).
- Indefinite License.

Description

There are several key issues that need to be determined for the Web Server architecture. The options available in the market to implement these technologies are listed below.

Web Server:

- Apache
- Netscape Server
- Microsoft Internet Information Server

CGI Technology

- Servlet/JSP
  - Tomcat (from Apache group)
  - JRun (from Allaire)
- Active Server Pages (available on NT only)
- NSAPI (C level API available for Netscape and Apache).
- ISAPI (C level API available for IIS and Apache)
- CGI (Perl/C etc.).

Database Connectivity

- JDBC.
- ODBC
- Native.

Database

- SQLServer
- Oracle
- Sybase
- Informix
- LDAP(??)

The overall proposed solution for eStream 1.0 WebServer release is:

Apache + Tomcat(for JSP/Servlet) + JDBC + SQLServer.

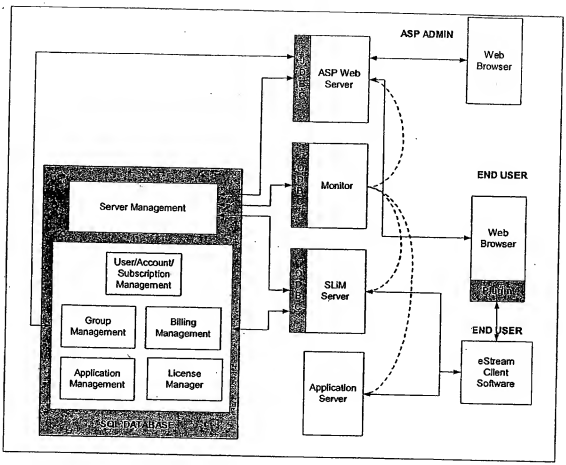
The reasons for choosing this combination for the servers are as follows:

1. JSP/Servlet is the only technology which is available for cross-platform and cross WebServer support.
2. We need to decide on a single web server to develop and test against for release 1.0. Apache is chosen to be the one as it is popular on Unix and NT platforms and it is freely available.
3. Tomcat(Apache group's reference implementation for JSP/Servlet specs) is the preferred CGI technology as it works well with Apache and all other web servers.
4. JDBC is preferred for database connectivity as its database neutral and works well with Java environment of Servlets.
5. SQLServer is the preferred database for release 1.0. This contains the scope for testing and deployment for eStream 1.0.

Since all other servers(App Server, SLM Server and Monitor) are C++ components, the following technology combination will be available for Database Access.

ODBC + SQLServer.

The data model for the eStream 1.0 database essentially consists of two high level components. The database deployment architecture is shown below:



Server Management Component: This component's primary responsibility is to manage the configuration, load and log information for a logical server in the system. The clients to this component are all the servers and administration manager. A detailed list of interfaces for this component is described in the interfaces section.

User/Account/Subscription Management: This component is responsible for maintaining the user account and subscription information for the system. The end user using the end user interface performs the updates to this component. Slim Server will access this component to validate subscriptions. A detailed list of interfaces for this component is described in the interfaces section.

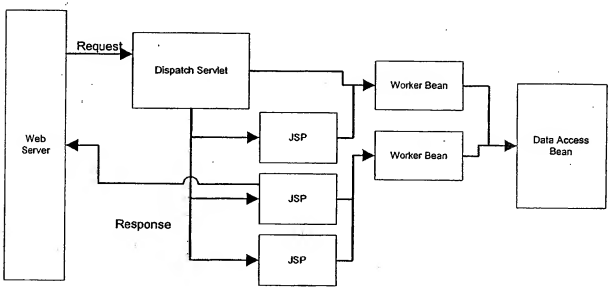
Group Management: This component is useful for managing groups of users. The group administrator can only perform updates to this component. A detailed list of interfaces for this component is described in the interfaces section.

Billing Management: This component's responsibility is to provide interfaces to an external billing system. A detailed list of interfaces for this component is described in the interfaces section.

Application Management: This component's responsibility is to provide application management interface. This component is accessible for updates only by the ASP administrator. A detailed list of interfaces for this component is described in the interfaces section.

License Manager: This component's responsibility is to manage the licenses. SLiM server will check out licenses from the license manager. A detailed list of interfaces for this component is described in the interfaces section.

The architecture for the Web Server extensions implementation is shown below:



The basic elements of this architecture are as follows:

1. Every request into the system goes through a dispatcher servlet. This servlet will perform initialization, initial validation of the request and miscellaneous checks before dispatching the request to a JSP page. A worker bean will responsible for performing the initialization. The processing of the incoming request is performed at this stage. The request is then dispatched to an appropriate JSP page.
2. The JSP page will invoke worker beans to access the dynamic data from the database via the Data Access Bean and the resultant page is sent back to the user.

This architecture is illustrated with the following example.

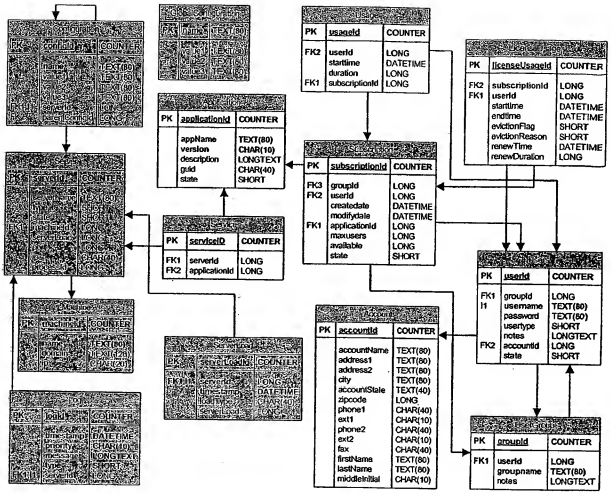
1. User sends in a request to update the username and password information in the database. Inputs are username, old password, new password.
2. The dispatch bean will call the user(worker) bean to:
    a. Validate the user's old password.
        i. The user worker bean will make a request to the data access bean to access the password for the user.
        ii. The two passwords are compared and the result is returned.
    b. If the password was valid then, update the new password.
        i. Call the data access bean to update the password in the database.
    c. Else return failure.
3. Based on the success or failure the dispatcher will dispatch the page request to the appropriate JSP page. (eg. error.jsp on failure and user.jsp on success).
4. The page will invoke the appropriate the worker bean (error bean or user bean) to obtain the dynamic data and send the response back to the user.

The salient features of this architecture are:

1. Presentation and processing logic is separate. Thus, the customer(ASP) can customize the look and the feel of the pages without impacting the processing logic as it is segregated.
2. The data access bean is separated from the worker beans, which are primarily responsible for the business logic. This allows us to change the data access layer (eg enabling LDAP access) in the future without impacting the system drastically.

Data type definitions

The central data structure for Web Server is the database model. The overall database model for user and subscription management is shown below.



The important features of this data model are:

Account: Table holding all the billing and contact information for a user or a group.

User: An end user in the system. A user can optionally belong to a group.

UserGroup: A group of users. One of the users in the group is designated as the group administrator. Each group has a unique account associated with it.

Application: This table contains the data about various applications in the supported by the ASP.

Subscription: This table contains entries for subscription items. A subscription item consists of user/group, application and license.

Usage: This table contains the runtime information for a system. SLiM server updates this table with access token usage data. A billing system may interface with this table to generate billing data. A reporting system may interface with this table to report on usage patterns.

LicenseUsage: This table is responsible for recording checked out licenses in the system.

The data model for storing the server related information is shown below:

PK: Primary Key for the table.

FK: Foreign Key. Used for relations between tables.

I1,I2.. : Index Columns.

Server: This table contains entries for each logical server in the system.

Machine: This table contains entries for each physical server in the system

Configuration: This table contains configuration entries for a given server. The configuration entries can be hierarchical in nature. Each configuration has the following format:

Name Value1 [Value2] [Value3] [ParentConfigId]

Load: This table maintains the historical and real-time load information for a given logical server in the system.

Service: The service table is used to record to map all the app servers and the corresponding applications that they serve.

Log: This table maintains the logs for a logical server in the system. The log messages saved here are "major" events in the logical server system. A detailed logs stored in a flat file on the physical machine containing the logical servers

Global Data Structures:

```
class DLLEXPORT Configuration
{
public:
        Cnfiguration() {mConfigId = -1; mServerId = -1; mParentConfigId = -1;};
        vrtual ~Configuration(){};
        int mConfigId;
        string mName;
```

```
        string mValue1;
        string mValue2;
        string mValue3;
        int mServerId;
        int mParentConfigId;
};

class DLLEXPORT GlobalConfiguration
{
public:
        GlobalConfiguration(){};
        virtual ~GlobalConfiguration(){};
        string mName;
        string mValue1;
        string mValue2;
        string mValue3;
};

class DLLEXPORT DBLog
{
private:
        long getTimeStamp();
public:
        DBLog();
        virtual ~DBLog();
        string mMessage;
        int mType;
        int mPriority;
        int mServerId;
        tm* mpTime; // Time stamp in ms since epoch.
};

class DLLEXPORT ServerLoad
{
public:
        ServerLoad(){mpTime=NULL; mLoadType=0; mServerLoad=0; mServerId=0;}
        virtual ~ServerLoad(){};
        tm* mpTime;
        int mLoadType;
        int mServerLoad;
        int mServerId;
};
```

The load data structure contains the following pieces of data:

- Server Id. Identifier for the server.

- Load: Response time in milliseconds.

For the Access Token and related data structures, please refer to the SLiM server Low Level Design Document. The interfaces below will discuss some of the API's based on the these data structures.

Interface definitions

The interfaces exposed by various sub-components are detailed below.

Server Management Component:

CreateServer

int CreateServer (ServerConfig* config)

```
Input:
   Server Configuration. (data structure defined
in the server configuration document.)
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
Errors:
   BAD CONFIGURATION
   NVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

UpdateServerConfig

Bool UpdateServerConfig(int serverId, String name, String value)

```
Input:
   Server Id
   Config name and value
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
Errors:
   BAD CONFIGURATION
```

```
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

AddMachine

Bool AddMachine(String name, String domain, String ip)

```
Input:
  Machine name,domain and ip.
Output:
  Success/Failure
Comments:
  Create a physical machine entry.
Errors:
  NO DATABASE CONNECTION
  UNKNOWN SQL ERROR
```

SetServerLog

Bool SetServerLog(int serverId, LogTuple log)

```
Input:
  Server Id
  Log tuple(data structure in the Logging
document.)
Output:
  Success/Failure
Comments:
  Add the log data for a server
Errors:
  INVALID SERVER ID
  NO DATABASE CONNECTION
  UNKNOWN SQL ERROR
```

GetServerLog

LogTuple[] GetServerLog(int serverId, int maxrows = 25)

```
Input:
  Server Id
  Maxrows: Maximum number of rows to be re-
turned.

Output:
```

```
   Array of Log tuples(data structure in the
Logging document.)
Comments:
   Get the log data for a server
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServers

ServerTuple[] GetServers()

```
Input:

Output:
   Array of Server tuples(data structure de-
fined above)
Comments:
   Get all the server information
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetServerState

Bool SetServerState (int serverId, short state)

```
Input:
   ServerId: Unique id for a server
   State: State information for a
server.(Defined in the server framework docu-
ment.)
Output:
   Bool True/False for success/failure.
Comments:
   Update the database with current state in-
formation for a specified server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerState: Obtain the last known state for a specified server

short GetServerState (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   State: State information for a server.
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerConfig: Obtain configuration information for a specified server

ServerConfig* GetServerConfig (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   ServerConfig*: State information for a
server. (ServerConfig data structure is defined
in the server configuration document).
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetLoadData:

void SetLoadData (int serverId, int Load)

```
Input:
   ServerId: Unique id for a server
```

```
   Load: Load for the server
Output:

Comments:
   Monitor may call this interface to persis-
tently store historical load data.  It is still
not clear if SLM and application servers will
store this directly themselves.
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerLoad:

void GetServerLoad (int serverId, , int maxrows = 25, int** Load)

```
Input:
   ServerId: Unique id for a server
   maxrows: Maximum number of rows to be re-
turned.Default is 25.
Output:
   Load: Load for the server
Comments:
   Obtain server component load information to
manage load balance.
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

FlushLoadData:

void FlushLoadData (<tuples> LoadData)

```
Input:
   LoadData tuples containing <server id,
server load> values.
Output:

Comments:
```

```
  Used to flush aggregated load data to the
databa
Errors:
  INVALID SERVER ID
  NO DATABASE CONNECTION
  UNKNOWN SQL ERROR
```

User/Account/Subscription Management Component

CreateUser. This API is used to create user record in the system. Arguments will be Username, Password.

Bool CreateUser(String username, String password)

ValidateUser. This API is used to validate user record in the system. Arguments will be Username, Password.

Bool ValidateUser(String username, String password)

CreateAccount. This API is used to create account records in the system. Arguments will be billing address, credit card information etc.

Bool CreateAccount(int userid, <Account Information>couple[])
Input:
Username associated with the account.
An array of names and values for the account.

AddSubscription. This API is used by the end users/group administrators to subscribe to applications.

Bool AddSubscription(int userid, <Subscription Information>couple[])
Input: An array of names and values for the subscription.

UpdatePassword. Used to change user information. Password, username etc.

Bool UpdatePassword(int userid, String old-password, String new-password);

UpdateAccount. Used to update the account information. Billing Address etc.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

UpdateSubscription. Used to add additional time to a subscription.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

GetUserRecord. Used to get current user configuration.

Couple[] GetUserRecord (int userid)

GetAccountRecord. Used to get current account configuration for a user.

Couple[] GetAccountRecord(int userid)

GetSubscriptionRecords. Used to get to subscription records in a database. End user may just want to verify what they are subscribed to.

Couple[][] GetSubscriptionRecords(int userid)

Output: An array of array of couples containing the subscription information for a given user.

DeleteUser. Used to delete users who are no longer valid in the system. Typically called by the ASP admin.

Bool DeleteUser(int userid)

DeleteAccount. Used to delete un-used accounts.

Bool DeleteAccount(int accountId)

DeleteSubscription. Used by the ASP admin to remove subscriptions.

Bool DeleteSubscription(int subscriptionId)

Group Management Component

CreateGroup. This API is responsible for creating group accounts in the database. Called by the group admin user.

Bool CreateGroup(String groupName, String admin, String notes)

AddUserToGroup. Adds a user to a group.

Bool AddUserToGroup(int groupid, int userid)

DeleteUserFromGroup. Removes a user from a group.

Bool DeleteUserFromGroup(int groupid, int userid)

GetActiveSessions. Gets the active sessions for a group.

Couple[][] GetActiveSessions(int groupid)

Output: An array of array of couples containing the following information for each active session in the system

Username
LicenseId
StartTime
EndTime
Subscription

DeleteGroup(int groupid); //Deletes the group

Couple[][]GetGroupUsers(int groupid); // Gets all the users for a given group.

Licensing Component

CheckoutLicense: Checks out a license.

int CheckOutLicense(int subscriptionId, long* pStartTime, long* pStopTime)

Inputs:

SubscriptionId: Subscription id of the user.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE

NO DATABASE CONNECTION
UNKNOWN SQL ERROR

RefreshLicense: Refreshes a license.

int RefreshLicense(int LicenseUsageId, long* pStartTime, long* pStopTime)

Inputs:

LicenseUsageId: License usage id.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
EVICTION

CheckinLicense: Check in a license

Bool CheckInLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Success/Failure

Comments:

Errors:

NO DATABASE CONNECTION
UNKNOWN SQL ERROR

ValidateLicense: Validate that the user has a license checked out.

Bool ValidateLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Yes/No.

Comments:

Errors:

```
INVALID USER
INVALID SUBSCRIPTION
INVALID LICENSE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

DBAcquireAccessToken

RPCReturnCodes DBAcquireAccessToken(long SubscriptionId, long* pAccessTokenId, string UserName, string Password, long* pStartTime, long* pStopTime, long* ApplicationId)

IN	SubscriptionId	Id of the subscription being used.
IN/OUT	pAccessTokenId	-1 if this is a first time access.
IN	UserName	Username string.
IN	PassWord	Encrypted Password
OUT	pStartTime	Start time for Access Token validity.
OUT	pStopTime	Stop time for Access Token validity.
IN/OUT	ApplicationId	Id of the application. –1 Default.
OUT	RPCReturnCodes	RPC Return codes.

Processing:

This is fairly complex function. The processing involved in this function call is:

- If this is the first access (ie *pAccessTokenId == -1) then **ValidateUser**
- If the ApplicationId is –1 then **GetAppId**
- If this is the first access (ie *pAccessTokenId == -1) then **CheckoutLicense**
- If this is a renewal request: **RefreshLicense**
- If there is a failure and it is due to eviction: **GetEvictionReason**

Errors:

```
#define     RPCR_USER_AUTH_FAILED
#define     RPCR_ACCESS_TOKEN_INVALID
#define     RPCR_ACCESS_TOKEN_EXPIRED
#define     RPCR_LICENSE_NOT_AVAILABLE
#define     RPCR_LICENSE_ALREADY_HELD
#define     RPCR_EVICTION_NOTICE
```

```
#define     RPCR_EVICTION_MUST_UPGRADE
#define     RPCR_EVICTION_END_MEMBERSHIP
#define     RPCR_EVICTION_NO_PAYMENT
```

DBReleaseAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:

- Update the Usage table with the appropriate information.
- Delete the LicenseUsage record.

Notes:

- We need a mechanism to release un-released access tokens. The way to do this would be to run a stored procedure at demand and at a predefined intervals to do this cleaning up.

EvictAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:

- Evicts an access token.

Billing Component

AddUsageRecord. Called by the SLM server when it releases an access token.

Bool AddUsageRecord(String username, int subscriptionId, date starttime, long duration).

GetUsageRecordsForUser. Used by external billing system.

Couple[][] GetUsageRecordsForUser(String username)

GetUsageRecordsForGroup Used by external billing system.

Couple[][]GetUsageRecordsForGroup (String groupName)

Application Management Component

AddApplication

int AddApplication(String appname, String version, String description)

Inputs:
Appaname: Application name.
Appversion: Application version
Description. Application description.
Outputs:
-1 for failure to add the application.
>0 otherwise. Application ID.
Comments:
Returns an app id for a newly added application.
Errors:
APPLICATION EXISTS
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(String appname, int version)

Inputs:
Appaname: Application name.
Appversion: Application version
Outputs:
-1 for failure to find the application.
>0 otherwise.
Comments:
Returns an app id for a given application.
Errors:
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(int SubscriptionId)

Inputs:
SubscriptionId
Outputs:
0 for failure to find the application.

>0 otherwise.

Comments:

Returns an app id for a given application.

Errors:

```
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetSubscribedApplicationIds

Int[]* GetSubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetUnsubscribedApplicationIds

Int[]* GetUnsubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids not subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetApplicationDetail

Couple[] GetApplicationDetail(int appid)

Inputs:

Application Id.

Outputs:

Array of couple for the app id containing:
{appname, appversion, description} values.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

Component design

We will discuss some complex scenarios in this section.

Subscription

Single New User

1. Create the user. **CreateUser.**
   a. If a user already exists, return error message and go back to 1.
2. Create the account for the user. **CreateAccount**
   a. Get the contact information from the user.
   b. Prompt to get the billing information. The user may decide to not give the billing information at this point.

Corporate group admin creating an account.

1. Create the admin user. **CreateUser.**
2. Create the group. **CreateGroup**
3. Create the account information for the group. **CreateAccount.**
   c. Get the contact information from the user.
   d. Prompt to get the billing information.
4. Add users to the group. **AddUserToGroup.**
   a. This method will automatically create the user if they do not already exist in the system.
   b. The list of users is accessible to the Group Admin by querying:
      i. Our database **GetUserRecords** OR
      ii. Some external database. Eg. LDAP directory.

Single User subscribing to an application

1. Validate the user. **ValidateUser**

2. Prompt to get the billing information if the billing information is not already present.
3. Get the list of un-subscribed applications. GetUnsubscribedApplications.
   a. **GetUnsubscribedApplicationIds.**
   b. For each app id returned, get the application details. **GetApplicationDetail**
4. For each additional application user wants to subscribe, call **AddSubscription**

SLiM server checking out an access token to use an application

1. Call **DBAcquireAccessToken.**

Releasing unreleased access tokens.

Unreleased access tokens will be release using a periodic stored procedure. The algorithm for stored procedure would be as follows:

1. For all records in the LicenseUsage table whose renewal time is past, call **DBReleaseLicense.** (Note that this can be a thread in SlimServer).

UI Rules

There are in general four kinds of UI components: textboxes, selects, checkboxes and radio buttons. Certain client side validations need to be applied to the entries of these components for each of the interfaces we develop in order to facilitate server development and processing. These validations are detailed as follows:

Textboxes

- String literal — this kind of textbox entry can only contain letters, numeric digits, underscores and spaces.
- Non-negative integer — this kind of textbox entry can only contain non-negative integers.
- IP address — this kind of textbox entry will consist of 4 textboxes, each with a char width of 3, and each of them must contain between 1 and 3 numeric digits (inclusive).
- Phone number — for the time being, we only care about US phone numbers. It will consist of 3 textboxes, the first two (char width 3) must contain 3 numeric digits each, and the last one (char width 4) must contain 4 numeric digits.
- Password — this kind of textbox entry must be at least 5 characters long.

Please note that they cannot be blank if they are required.

Selects

- Drop-down — at least one selection must have been made if required.
- List — at least one selection must have been made if required.
- Date — this will consist of 5 drop-down boxes, a selection must be made in each of them if a date is required, the default will be set to the current date and time on page load or reset.

Checkboxes

- At least one of a group must have been selected if required.

Radio Buttons

- One and only one of a group must have been selected if required.

Testing design

This document must have a discussion of how the component is to be tested. Some sub-sections could include:

Unit testing plans

The follwowing components will be unit tested:

ODBC connectivity dll for the SliM server and the Monitor. A simple C++ executable will be provided to test the SLiM server and the Monitor interfaces. The C++ executable will:

- establish connection to the database.
- simulate access token calls.
- simulate the monitor calls.

The Web servers' servlets and the JSP pages interact with the database using a set of java beans. Each of the bean will have a test interface. A simple JSP will be provided which will call all of the test interfaces for the beans. The test interface itself will be responsible to call all the interfaces that the bean provides in a predefined calling sequence.

The servlets will be unit tested using a set of html forms which will invoke the servlets.

Stress testing plans

Stress testing will be invoked using some external testing tool which can record HTTP traffic and replay the traffic for multiple users. Performix and LoadRunner are two possible choices. (There may be additional tools available).

Coverage testing plans

Coverage on the ODBC components will use the same mechanism as the rest of the server code. Pure Coverage may be used to achieve this goal.

Coverage on the Java components is an open issue. We need to investigate the appropriate tools for doing this testing.

Cross-component testing plans

The database is the central point for distribution of the data from Web server to the rest of the servers. Thus, creating the database data which can be used by Slim server and the App server will be a good source of cross component testing plan.

Upgrading/Supportability/Deployment design

We will be finally shipping just the java class files and JSP pages to the customer. It is assumed that the customer will have the appropriate web server to support JSP 1.1 and Servlet 2.2.

Open Issues

1. We have assumed that the JDBC implementation will come from Inet software. We may need to change to an alternate JDBC vendor based on pricing, quality etc.
2. Tomcat is decided to be the JSP/Servlet engine. Again, this is a freeware and may be replaced by a commercial version (Jrun from Allaire).
3. The web server will need to talk to the Monitor. The messaging component for this communication is not well defined yet.